RepoGraph: A Novel Semantic Code Exploration Tool for Python Repositories based on Knowledge Graphs and Deep Learning

Christopher Williams
School of Computer Science
University of St Andrews
St Andrews, UK
cjdw1@st-andrews.ac.uk

Rosa Filgueira
School of Computer Science
University of St Andrews
St Andrews, UK
rf208@st-andrews.ac.uk

***Abstract*—This work presents RepoGraph, an integrated semantic code exploration web tool that combines information extraction, knowledge graphs, and deep learning models. It offers new capabilities for software developers (from academia and industry) to represent and query Python repositories. Unlike existing tools, RepoGraph not only provides a novel search interface powered by deep learning techniques but also exposes the underlying features and representations of repositories to users. Additionally, it offers several interactive visualizations. We also introduce RepoPyOnto, a new ontology that captures the features of Python code repositories and is used by RepoGraph for representing the captured knowledge. Finally, we successfully evaluate RepoGraph against several criteria, including function summarization performance, the correctness and relevance of search results, as well as the processing time for constructing graphs of various sizes.**

***Index Terms*—Static Code Analysis, Code Exploration, Code Understanding, Semantic Code Search, Knowledge Graph, Function Summarization, Deep Learning Models, Transfer Learning**

I. Introduction

The concept of "Programming as Theory Building" by Peter Nau [1] emphasizes the importance of robust and comprehensible mental models in software development. Without such models, the dissolution of a programming team possessing the theoretical understanding of a program can lead to its demise. In today's academic and industry software development landscape, Naur's theory holds even greater significance [2]. With millions of software developers worldwide, it becomes crucial to equip software teams with effective tools that facilitate the construction and maintenance of mental models to comprehend how their software operates, decreasing debugging efforts, and improving the knowledge sharing within the team.

The Semantic Web movement has popularized knowledge graphs as a valuable tool for storing and representing domain-specific knowledge [3]. By combining a well-defined ontology and a flexible graph database system, knowledge graphs have the potential to serve as a human-understandable foundation for future 'code intelligence' tools [4]. Advances in machine learning techniques have further contributed to the realization of 'semantic search' [5], which leverages domain understanding to enhance search capabilities. 'Semantic code search' [6] aims to apply these techniques to search codebases. Given that knowledge graphs can encode our understanding of software repositories, semantic code search represents a complementary technique in this context.

Numerous 'code intelligence' tools have emerged in recent years, offering analysis and indexing of software repositories, such as KG4Py [7] or CodeClimate [1]. While these tools provide search interfaces, they often lack transparency by not exposing their underlying models or representations to users. To address these limitations, this work introduces `RepoGraph`, an innovative semantic code-search web tool. `RepoGraph` harnesses the power of knowledge graphs to represent and query Python code repositories, providing a solid foundation for semantic code search techniques. Importantly, `RepoGraph` also promotes transparency and understanding by exposing repository representations to users. The main contributions of this work are:

- A new ontology, `RepoPyOnto`, that captures the features of a Python repository extracted with `inspect4py`.
- A novel methodology to generate automatically knowledge graphs of Python code repositories that conform to `RepoPyOnto` ontology.
- Augmentation of the information previously captured in the knowledge graphs with several deep learning models.
- A new semantic code search mechanism that utilises knowledge graphs and deep learning transformers.
- A new user interface to allow users to create (and store) automatically knowledge graphs from single or multiple python repositories, visualize their main features, their possible issues, as well as perform semantic code searches on them.

By facilitating comprehensive code understanding and promoting transparency, `RepoGraph` addresses the crucial need for effective tools that enhance software teams' productivity and enable seamless collaboration in both industry and research settings. The remainder of the paper is structured as

[1]https://github.com/codeclimate/codeclimate

follows. Section II presents background on `inspect4py`. Section III details the features of the `RepoPyOnto` ontology. Section IV gives an overview of the main features of `RepoGraph`. Section V gives a detailed introduction of the core services implemented in `RepoGraph`, along with the deep learning models employed at this work. Section VI introduces the different components of the `RepoGraph` User Interface. Section VII, describes the evaluations performed to validate `RepoGraph`. Finally, section VIII summarises related work, and section IX concludes with a summary of achievements and future work.

II. INSPECT4PY

`inspect4py` [8], a static code analysis tool, is a project of significant relevance to this work. It extracts two main types of features from a Python code repository: 'Software understanding features' and 'Code features'. The 'Software understanding features' aim to facilitate the adoption of a software package and include:

- **Class and function metadata and documentation**: Extracts information such as name, inherited classes, documentation, methods, function arguments, returned values, relevant variables, and detects nested functions.
- **Requirements**: Provides a list of required packages and their corresponding versions.
- **Dependencies**: Lists the internal and external modules used by the target software.
- **Tests**: Identifies files used for testing the software's functionality.
- **Software invocation**: Ranks the different alternatives to run the software component based on relevance.
- **Main software type**: Estimates whether the target software is a package, library, service, or series of scripts.

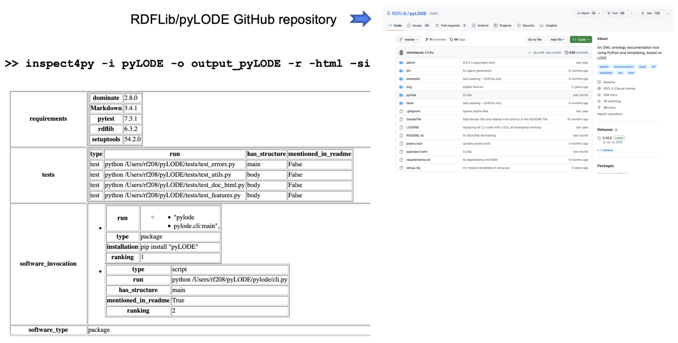
On the other hand, the 'Code features' aim to characterize the code from different perspectives and include:

- **File metadata**: Tracks included classes, methods, dependencies, presence of a main method, and the existence of a file body.
- **Control flow graph**: Retrieves the control flow representation of each file as a text file and figure, providing insights into the program's possible execution paths.
- **Call list**: Extracts a list of all involved functions for each function, method or code body.
- **File hierarchy**: Records the organization and grouping of files within the software repository.

As a result, when using `inspect4py`, a folder is generated containing a summary file that includes the features selected by users. For instance, Figure 1 shows a subset of the features extracted by `inspect4py` from the PyLODE[2] repository.

In this work we have used `inspect4py` for allowing `RepoGraph` to extract the features of a given repository, and to map them later into a knowledge graph. Section V-A explain this process in detail. Furthermore, `RepoGraph` augments the information extracted from `inspect4py` with: deep

[2]https://github.com/RDFLib/pyLODE



```
>> inspect4py -i pyLODE -o output_pyLODE -r -html -si
```

requirements	**dominate** 2.8.0; **Markdown** 3.4.1; **pytest** 7.3.1; **rdflib** 6.3.2; **setuptools** 54.2.0
tests	see below
software_invocation	see below
software_type	package

tests:

type	run	has_structure	mentioned_in_readme
test	python /Users/rf208/pyLODE/tests/test_errors.py	main	False
test	python /Users/rf208/pyLODE/tests/test_utils.py	body	False
test	python /Users/rf208/pyLODE/tests/test_doc_html.py	body	False
test	python /Users/rf208/pyLODE/tests/test_features.py	body	False

software_invocation:

run	"pylode; pylode.cli:main",
type	package
installation	pip install "pyLODE"
ranking	1

type	script
run	python /Users/rf208/pyLODE/pylode/cli.py
has_structure	main
mentioned_in_readme	True
ranking	2

Fig. 1: Features of pyLODE, an OWL ontology documentation package, extracted with `inspect4py`.

learning techniques to generate function summarizations (see Section V-D); code call graph generation (see Section V-A); and repository issues detection (See Sections V-B and V-E3).

III. REPOPYONTO ONTOLOGY

In this work, the `RepoPyOnto` ontology was developed to represent entities and relationships found in Python code. This ontology captures the core entities extracted by `inspect4py` and incorporates important design choices. The `Function` entity (see Table I), which represents pure functions and class methods, is a crucial entity type, distinguished by the type property (function or method). Relationships such as `HasMethod` and `HasClass` (see Table II) connect this entity to `Class` and `Module` entities. It is worth noting that the summarization property is incorporated into the `Docstring` entity instead of creating a separate `Summarization` entity. This decision aims to simplify the knowledge graphs and limit them to entities extracted by `inspect4py`, while the function summarization is provided by `RepoGraph` (see Section V-D). The data model of `RepoPyOnto` can be accessed at [3].

IV. REPOGRAPH OVERVIEW

In this work, we present `RepoGraph`[4], a novel React-based web application to explore Python code repositories by employing knowledge graphs and deep learning models. `RepoGraph` allows users to quickly generate automatically knowledge graphs that represent single or multiple repositories, and to interact with them. Neo4j [5] is the graph technology chosen to generate our knowledge graphs, since is considered one of the most popular and widely-used graph database implementation since 2013.

To enhance user interaction in `RepoGraph`, we have developed convenient abstractions that eliminate the need for users to write Cypher queries, which is Neo4j's graph query language [9]. The novel `RepoGraph` functionalities include:

1) Automatic generation of knowledge graphs from one or more Python code repositories.

[3]https://drive.google.com/file/d/1qPF_zQoJ0vtjEDUCPJbf1cm5SgLR92ai/view?usp=share_link
[4]Availabe at https://github.com/WilliamsCJ/repograph
[5]Neo4j: https://neo4j.com/

Entity	Properties
Repository	name, full_name, description, type, ...
README	path, content
License	text,license_type, confidence
Directory	name, path, parent_path
Package	name, canonical_name, parent_package, path, parent_path, external, inferred
Module	name, canonical_name, path, parent_path, extension, is_test, inferred
Class	name, canonical_name, min_line_number, max_line_number
Function	name, type, builtin, canonical_name, source_code, ast, min_line_number, max_line_number, inferred
Variable	name, canonical_name, type, inferred
Argument	name, type
Return Value	name, type
Body	source_code
Docstring	short_description, long_description, summarization
Docstring Argument	name, type, description, is_optional, default
Docstring Return Value	name,type, description, is_generator
Docstring Raises	description, type

TABLE I: `RepoPyOnto` entity set.

Relationship	Source	Destination
Requires	Repository	Package
Contains	Repository, Directory, Package, Module	Directory, Module, Package, README, Function, Class, Body, Variable
Imports	Module	Module, Package, Class, Function, Variable
HasMethod	Class	Function
HasFunction	Module	Function
Extends	Class	Class
HasArgument	Function	Argument
LicensedBy	Repository	License
Documents	Docstring	Function, Class
Describes	Docstring	DocstringArgument, DocstringRaises, DocstringReturnValue
Calls	Module, Function	Function, Class, Module

TABLE II: `RepoPyOnto` relationship set.

2) Browse knowledge graphs and select a specific graph for further inspection.
3) View summary statistics about knowledge graphs.
4) Inspect a visualisation of an entire knowledge graph.
5) Execute semantic code search queries.
6) Inspect semantic code search results in detail with summarization information, and interactive visualisations.
7) Execute pre-determined queries using an interface that does not require knowledge of the Cypher language.
8) Browse possible 'issues' detected in repositories code.
9) Inspect individual 'issues' in more detail.
10) Delete knowledge graphs.

These functionalities are facilitated by a core set of services located at the back-end of `RepoGraph`, which will be detailed in the following section. The user interface, discussed in Section VI, also contributes to providing these functionalities.

V. RepoGraph Services

A `RepoGraph` service is an area of functionality that operates as a self-contained unit. In `RepoGraph` we have five core services (see TableIII). To provide complex functionality, interaction occurs between those services (full architecture available at [6]). For example, the *Search Service* implements semantic code search functionality. However, to access the data, this service call functions defined in the *Graph Service*.

Service	Purpose
Build	Knowledge graph construction from a single or multiple repositories. Section V-A.
Graph	Management and querying of a graph. Section V-B.
Metadata	Graph metadata management, including capturing and storing all graph associated metadata. Section V-C
Summarization	Code summarizations generation. Section V-D
Search	Execution of 'semantic code' and 'predefined' search queries. Section V-E

TABLE III: Reposrah Core Services

Behind each of these services we have developed our novel methods and heuristics for enabling semantic code exploration of Python repositories. Those are explained bellow.

A. Build Service

This service is responsible for the generation of knowledge graphs, which relies on `inspect4py` (see section II) to extract features from the supplied Python repositories. As it was mentioned before, Neo4j was selected as the graph technology to generate our knowledge graphs, while `RepoPyOnto` (see Section III) is used as the graph ontology. Note that each knowledge graph contains either single or multiple repositories - depending on the number of repository zip files uploaded while creating the graph (see Figure 5).

Furthermore, a core feature of the knowledge graphs generated by `RepoGraph` is that they are able to capture the relationships that describe the interactions between components (packages, functions, modules, classes, etc). This allows users to understand the control flow of their software. To generate these relationships, we have developed within the *Build Service* a new heuristic to parse the dependencies and call lists generated by `inspect4py`. The heuristic applies the Deep-First-Search (DFS) algorithm to create a *code call graph* (either for an individual function or for the whole repository) representing not only the calls that are made by each function, as well the calls that each of those calls make, which can imply other functions.

B. Graph Service

The *Graph Service* provides the set of models to represent the *entities* and *relationships* defined in `RepoPyOnto`. As well as a set of methods for providing the basic functionality for querying and writing data to a knowledge graph. Furthermore, since `RepoGraph` enables users to create and

[6]https://drive.google.com/file/d/1NT8sziYnDaw2enm3PSq231pdF016XLuw/view?usp=share_link

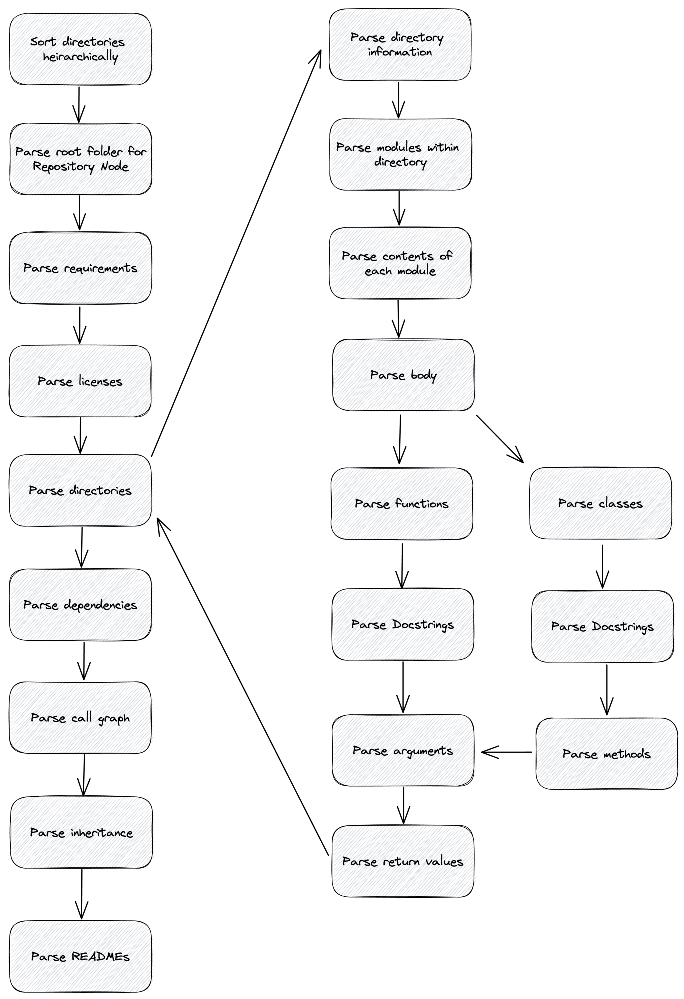


Fig. 2: *Build Service* pipeline. The information is parsed from `inspect4py`, and augmented in order to obtain the final knowledge graph.

store several knowledge graphs, these are stored in a *Graph Repository*. The *Graph Service* controls access to the *Graph Repository*, providing a broad access to read data from, and execute queries on, graphs stored in the repository.

`RepoGraph` analyses potential issues regarding with code repositories, at two levels: *repository*, and *docstring*. The *Graph Service* is responsible for detecting the issues at repository level ('Repository Issues'), while the *Search Service* (see Section V-E3) is responsible for issues at docstring level ('Docstring-related Issues'). `RepoGraph` detects two 'repository issues'. The first one is *Circular dependencies*, when two or more modules depend on each other. And the second one is *Missing dependencies*, when there are packages imported by the repository source code, but those are not explicitly. Therefore, the *Graph Service* analyses specifically the code call graph, requirements and dependencies information from knowledge graphs to detect *cyclical dependencies* and *missing dependencies* issues.

C. Metadata Service

`RepoGraph` captures metadata information for each knowledge graph storing them in the *Metadata Repository*. This includes the graph display name, the unique graph name, the timestamp when the build process started, along with the status: ACTIVE (once it is created), or PENDING (while the graph is being created). This information is capture and accessible to users by the user interface (see Figures 5, 6).

D. Summarization Service

`RepoGraph` also augments the knowledge previously extracted (see Section V-A) from Python repositories thanks to the *Sumarization Service*, which adds additional summarizations of functions using pre-trained transformer models.

Function summarization aims to summarize a function-level code snippet into English descriptions [10]. We utilise functions summaries as the foundation for the semantic code search, as well for detecting possible docstrings-related issues provided by the *Search Service* (see Section V-E).

In the *sumarization service*, we have evaluated three transformer models: `codet5-base-multi-sum` [7]; `codet5-small-code-summarization-python`[8]; and a new fine-tuned model, `codet5-base-python-sum`, model developed with this work [9]. While the first two models are multi-language checkpoints (that is not explicitly fine-tuned for Python), `codet5-base-python-sum` has been fine-tuned using using the Python 'code-to-text' split of the CodeXGLUE dataset [4].

Results of this evaluation are detailed in Section VII-A. Since `codet5-base-multi-sum` is the model that gives us the best performance (see Figure 11a), we have selected it as the model to use in the *Summarization Service*. Note that we have implemented this service in such a way, that is trivial to interchange this model, by another one that gives even better performance results.

The pipeline behind the *Summarization Service* has the following steps. First, each function source code is 'cleaned' removing their docstrings to reduce their influence in the summarization model. After cleaning, the source code is input into an instance of the RoBERTa tokenizer [10], which generates a set of 'input IDs' representing the source code. These 'input IDs' are passed into the model to generate a corresponding set of integer 'IDs'. Finally, the model decodes these IDs to produce a function summarization using `codet5-base-multi-sum`. This process is detailed in Figure 3. Resulting summarizations are added back into the knowledge graph.

E. Search Service

The *Search Service* implements functions related to search, including both semantic code search and pre-defined Cypher

[7]Fine-tuned checkpoint of the `Code T5` model trained on the CodeSearchNet data and provided at https://huggingface.co/Salesforce/codet5-base-multi-sum

[8]Fine-tuned checkpoing from the `codet5-small` model and provided at https://huggingface.co/stmnk/codet5-small-code-summarization-python

[9]Fine-tuned model: https://huggingface.co/cjwilliams/codet5-base-python-sum/settings

[10]https://huggingface.co/docs/transformers/model_doc/roberta

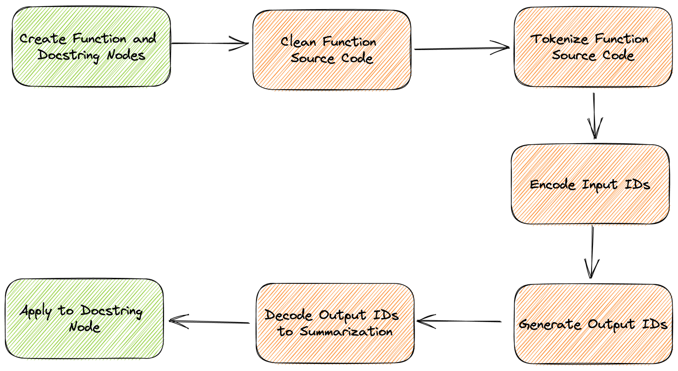

Create Function and Docstring Nodes
Clean Function Source Code
Tokenize Function Source Code
Encode Input IDs
Generate Output IDs
Decode Output IDs to Summarization
Apply to Docstring Node

Fig. 3: *Summarization Service* pipeline. Green indicates actions performed by the *Build Service* , and orange indicates actions performed by the *Summarization Service.*

search queries, and other functionalities that utilise cosine similarity to analyse docstrings (see Section V-E3). These are introduced as follows.

1) Semantic Code Search: Semantic code search is the task of retrieving relevant code given a natural language query [11]. This task can provide an intuitive way for users to navigate large code bases. This service utilises the summarizations generated by the *Summarization Service* for this task, as it shows in Figure 4.

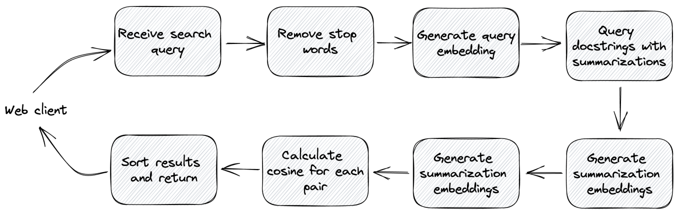

Receive search query
Remove stop words
Generate query embedding
Query docstrings with summarizations
Web client
Sort results and return
Calculate cosine for each pair
Generate summarization embeddings
Generate summarization embeddings

Fig. 4: *Semantic code search* pipeline. When calculating cosine similarities, a pair is defined as the natural language search query embedding and a given summarization embedding.

Embeddings are generated for all function summarizations (and stored), as well for each of the natural language queries performed by the users (e.g. *functions that call RDFLib*). And later, the cosine similarities scores are calculated between the summarization embeddings and the natural language query embedding. Prior calculating a natural language query embedding, the text is 'cleaned' by removing stop words using NLTK[11] - NLTK's stopword list[12].Later, embeddings are generated using, `multi-qa-distilbert-cos-v1`[13]. This model utilises the Sentence-BERT model to 'derive semantically meaningful sentence embeddings' [12], and it has been selected because it has been trained with data from the StackExchange website [14], which includes significant content

[11]NLTK: https://www.nltk.org/

[12]NLTK Stopwords: https://gist.github.com/sebleier/554280

[13]Hugging Face: https://huggingface.co/sentence-transformers/multi-qa-distilbert-cos-v1

[14]https://stackexchange.com/

Pre-defined Query	**Used in**
Select the requirements of a given repository	Cypher Search
Select the READMEs of a given repository	Cypher Search
Select the metadata of a given repository	Cypher Search
Select the license of a given repository	Cypher Search
Select the docstring (long and short) of a given repository	Cypher Search Semantic Code Search
Select the function summarizations of a given repository	Cypher Search Semantic Code Search
Select the file names of a given repository	Cypher Search
Select the function and class names for a given repository	Cypher Search Semantic Code Search
Select the source code for a given repository	Cypher Search Semantic Code Search
Select the call graph for a given repository	Semantic Code Search Summarization

TABLE IV: List of pre-defined queries used by *Search* and *Function Summarization* services

related to Python. Interchanging this model by another one, will be also a trivial operation in `RepoGraph`.

The *semantic code search* service is currently limited to functions and methods. Note that the summarisation embeddings are generated only once (the first time that a semantic code search is performed), storing them for future semantic code searches.

2) Cypher Search: As an alternative to semantic code search (limited to functions and methods), the *search service* allows also for knowledge graphs to be searched using a predetermined set of Cypher queries (see Table IV). These queries cover most aspects of the knowledge graph and should be sufficient for most use cases. Note, that not only the *Search service* uses these for implementing the *Cypher Search* and *Semantic Code Search* functionalities, as well by *Function Summarization Service*.

The *Search Service* utilise the *Graph Service* to execute these pre-defined queries and allow repository-level filtering in multi-repository graphs. This benefits the `RepoGraph` architecture and allows for further pre-defined queries extension.

3) Docstring-related Issues: The *Search service* also provides some functionality related to detecting 'Repository Issues' (previously defined in Section Section V-B) - namely 'Docstrings-Related Issues':

- *Possible incorrect docstring*: when a function or a method includes an incorrect docstring.
- *Missing docstring*: when a function or method has missing its docstring.

This service analyse the knowledge graphs functions and methods docstrings to detect *possible incorrect docstrings*. It employs the `multi-qa-distilbert-cos-v1` model (introduced at Section V-E1) to compare functions original docstrings (the ones included by the developer) with the docstrings automatically generated by the *Summarization Service*. Our hypothesis is that a low cosine similarity may indicate that the original docstring is incorrect, as it may have been copied from another function or become out-of-date. The low cosine score is used to calculate the 'possible incorrect docstrings' issue metric and we have set up the threshold to 0.25. This value was determined from our analysis of the similarity between summarizations generated by `codet5-base-multi-sum` and the docstrings they were generated from in the CodeXGlue [4] dataset.

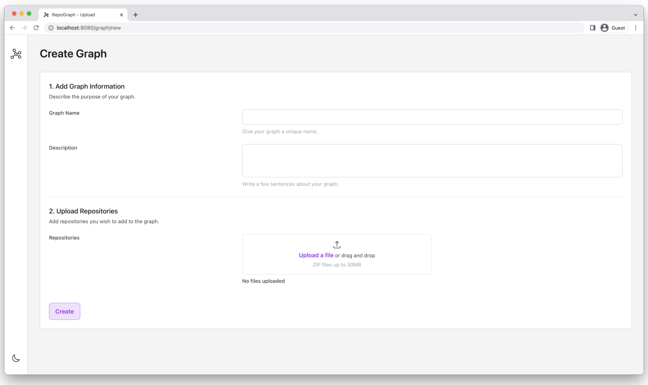
Furthermore, during the process of determining if docstrings are possibly incorrect, the *search service* also detects which functions have missing their corresponding Docstring. This is used to calculate the *missing docstrings* issue metric.

VI. RepoGraph User Interface

The User Interface (UI) is designed to provide a simple tool for users to utilise the knowledge graphs generated by the system to further their understanding of their repositories. The UI is designed with three key sections: Home, Graph, Search. Each of them are detailed bellow.

A. Home Section

The Home section has the *Create graph* page where users can upload repositories to generate new graphs (see Figure 5). As we mentioned in Section V-A, users can create single or multi-repository graphs (repositories need to be zipped to be uploaded), name them, and also store an associated description. This information is stored in the *Metadata Repository* by the *Metadata Service* (see Section V-C).


RepoGraph - Upload
localhost:8080/graph/new
Guest

Create Graph

1. Add Graph Information
Describe the purpose of your graph.
Graph Name
Give your graph a unique name.
Description
Write a few sentences about your graph.

2. Upload Repositories
Add repositories you wish to add to the graph.
Repositories
Upload a file or drag and drop
ZIP files up to 30MB
No files uploaded

Create

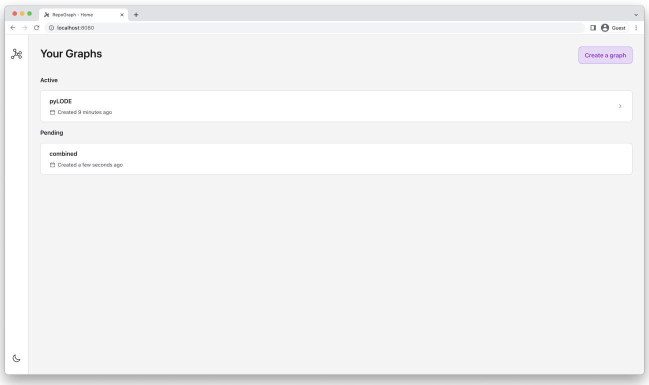
Fig. 5: *Create Graph* page from `RepoGraph` Home section .

To mitigate long waits for creating users graphs, we have employed the optimistic acceptance and polling approach is used, in which `RepoGraph` accepts an upload before it has been fully processed. The UI then waits for the server to finish processing by polling in the background before making it accessible to the user.

The Home section has also *Your Graphs* page, which provides a listing of generated graphs for users to interact with. Given possible long build times, `RepoGraph` captures the status of a graph (information stored at the *Metadata Repository*), and visualize it in in this section (see Figure 6). Upon upload, a graph is allocated the PENDING status and is only upgraded to ACTIVE once the build process has been completed successfully.

B. Graph Section

The Graph section is also composed of two pages. A *Summary* page (see Figure 7) that contains key information


RepoGraph - Home
localhost:8080
Guest

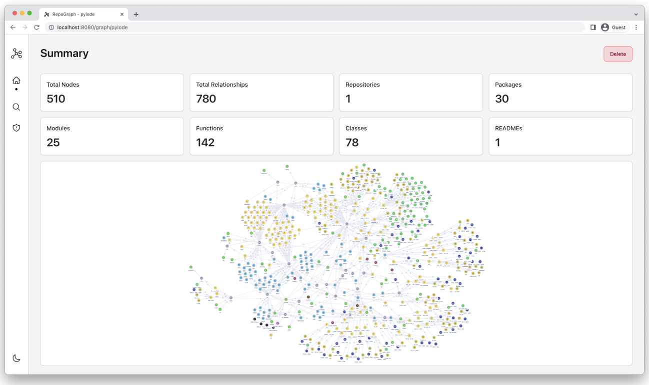
Your Graphs
Create a graph

Active
pyLODE
Created 9 minutes ago

Pending
combined
Created a few seconds ago

Fig. 6: Pending and active graphs on *Your Graphs* page. The active graph corresponds to pyLODE repository, while the pending correspond to a graph with multiple repositories.

on a graph (such as the number of files, repositories, etc.). This page also contains the visualization of the graph, which is interactive. So users can zoom in/out of them, obtaining nodes and relationship details. This information is retrieved from the *Graph Service*.

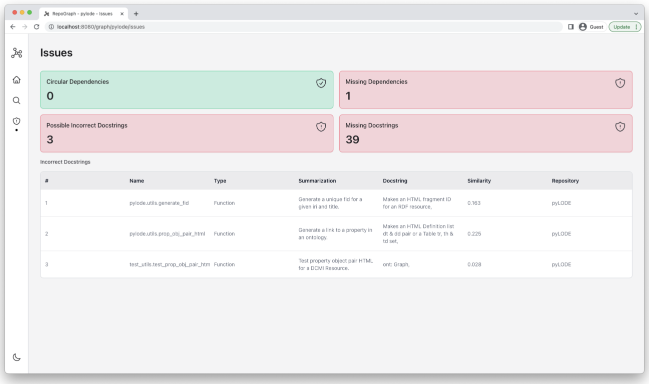

RepoGraph - pylode
localhost:8080/graph/pylode
Guest

Summary
Delete

Total Nodes	Total Relationships	Repositories	Packages
510	780	1	30
Modules	**Functions**	**Classes**	**READMEs**
25	142	78	1

Fig. 7: pyLODE repository graph *Summary* page showing statistics and knowledge graph visualisation.

An *Issues* page (see Figure 8) where users are alerted to potential problems with the repositories contained in the graph (these metrics leverage elements of graph theory, such as cycles, and machine learning methods), as it was introduced in Section V. These issues are calculated by *Graph* (Circular Dependencies and Missing Dependencies) and *Search* services (Possible Incorrect Docstrings and Missing Docstrings). Figure 8 shows an example of the *Issues* page for detecting the possible issues for pyLODE repository. Each issue is clickable in this, and more detailed information is available. In this Figure, we can see the detailed information of ‘Possible Incorrect docstrings’. For each ‘possible’ incorrect docstring, this Figure shows the function’s name, type (function or method), generated summarization (by the *Summarization Service*, original docstring (included by the source code), the similarity score (calculated by the *Search service*), and the repository name.


RepoGraph - pylode - Issues

localhost:8080/graph/pylode/issues

Issues

Circular Dependencies 0

Missing Dependencies 1

Possible Incorrect Docstrings 3

Missing Docstrings 39

Incorrect Docstrings

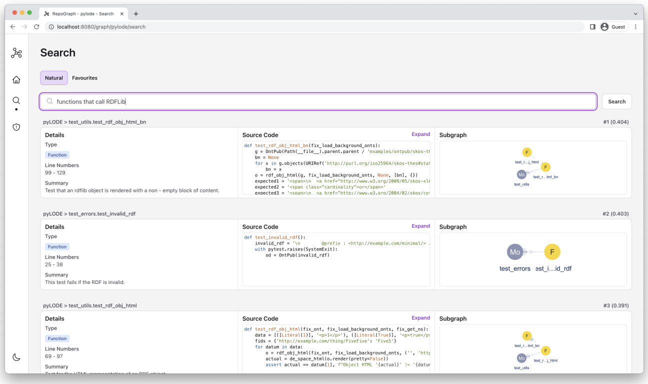
#	Name	Type	Summarization	Docstring	Similarity	Repository
1	pylode.utils.generate_fid	Function	Generate a unique fid for a given iri and title.	Makes an HTML fragment ID for an RDF resource,	0.163	pyLODE
2	pylode.utils.prop_obj_pair_html	Function	Generate a link to a property in an ontology.	Makes an HTML Definition list dt & dd pair or a Table tr, th & td set,	0.225	pyLODE
3	test_utils.test_prop_obj_pair_htm	Function	Test property object pair HTML for a DCMI Resource.	ont: Graph,	0.028	pyLODE

Fig. 8: Overview of the issues founded for pyLODE repository. The 'Possible Incorrect Docstrings' have been been clicked for further inspection.

C. Search Section

This section is where a user can use both semantic code search (see Figure 9) and pre-defined queries (see Figure 10a) to explore a graph. The *Search* page has have two tabs: a *Natural* tab for semantic code search queries; and a *Favourites* tab for pre-defined queries. Both tabs uses the *Search Service* introduced at Section V-E.

Figure 9 shows an example of the *Natural tab* to search functions that are more similar to the natural language search query *functions that call RDFLib* in the pyLODE graph. The obtained results in this page are sorted by their similarity score. Here, we can see the most similar functions, their source code, and their code call graph (which is also interactive).


RepoGraph - pylode - Search

localhost:8080/graph/pylode/search

Search

Natural | Favourites

functions that call RDFLib

pyLODE > test_utils.test_rdf_obj_html_bn — #1 (0.404)

Details — Type: Function — Line Numbers: 99 - 129 — Summary: Test that an rdflib object is rendered with a non - empty block of content.

Source Code

```
def test_rdf_obj_html_bn(fix_load_background_onts):
    g = OntPub(Path(__file__).parent.parent / 'examples/ontpub/skos-tl
    bn = None
    for x in g.objects(URIRef('http://purl.org/iso25964/skos-thes#stat
        bn = x
    o = rdf_obj_html(g, fix_load_background_onts, None, [bn], {})
    expected1 = '<span>\n  <a href="http://www.w3.org/2008/05/skos-xl
    expected2 = '<span class="cardinality">or</span>'
    expected3 = '<span>\n  <a href="http://www.w3.org/2004/02/skos/co
```

pyLODE > test_errors.test_invalid_rdf — #2 (0.403)

Details — Type: Function — Line Numbers: 25 - 38 — Summary: This test fails if the RDF is invalid.

Source Code

```
def test_invalid_rdf():
    invalid_rdf = '\n        @prefix : <http://example.com/minimal/> .
    with pytest.raises(SystemExit):
        od = OntPub(invalid_rdf)
```

pyLODE > test_utils.test_rdf_obj_html — #3 (0.391)

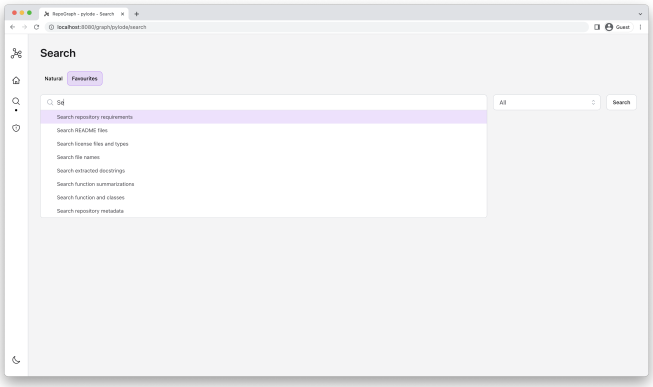
Details — Type: Function — Line Numbers: 69 - 97 — Summary

Source Code

```
def test_rdf_obj_html(fix_ont, fix_load_background_onts, fix_get_ns):
    data = [([Literal(1)], '<p>1</p>'), ([Literal(True)], '<p>true</p
    fids = {'http://example.com/thing/FiveFive': 'Five5'}
    for datum in data:
        o = rdf_obj_html(fix_ont, fix_load_background_onts, ('', 'http
        actual = de_space_html(o.render(pretty=False))
        assert actual == datum[1], f"Object HTML '{actual}' != '{datum
```

Fig. 9: Executing a semantic code search in RepoGraph.

The available pre-defined search options under the *Favourite* tab are shown in Figure 10a. Each option corresponds to one of the pre-defined Cypher queries introduced earlier in Table IV. An example of those is available in Figure 10b, in which we can see the results of executing *Search function summarizations* query for pyLODE graph.


RepoGraph - pylode - Search

localhost:8080/graph/pylode/search

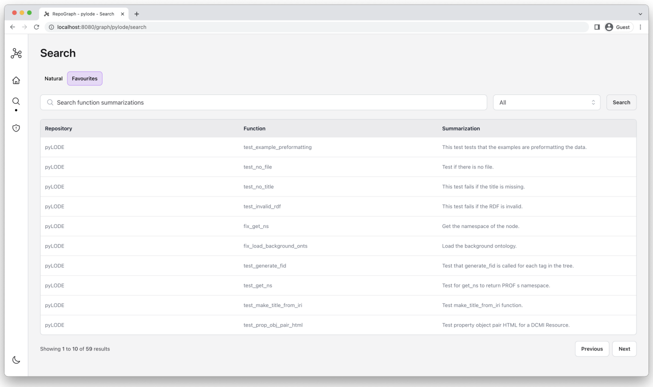
Search

Natural | Favourites

Se

- Search repository requirements
- Search README files
- Search license files and types
- Search file names
- Search extracted docstrings
- Search function summarizations
- Search function and classes
- Search repository metadata

(a) Favourite search bar showing available options.


RepoGraph - pylode - Search

localhost:8080/graph/pylode/search

Search

Natural | Favourites

Search function summarizations

Repository	Function	Summarization
pyLODE	test_example_preformatting	This test tests that the examples are preformatting the data.
pyLODE	test_no_file	Test if there is no file.
pyLODE	test_no_title	This test fails if the title is missing.
pyLODE	test_invalid_rdf	This test fails if the RDF is invalid.
pyLODE	fix_get_ns	Get the namespace of the node.
pyLODE	fix_load_background_onts	Load the background ontology.
pyLODE	test_generate_fid	Test that generate_fid is called for each tag in the tree.
pyLODE	test_get_ns	Test for get_ns to return PROF s namespace.
pyLODE	test_make_title_from_iri	Test make_title_from_iri function.
pyLODE	test_prop_obj_pair_html	Test property object pair HTML for a DCMI Resource.

Showing 1 to 10 of 59 results

(b) Executing *Search function summarization* query.

Fig. 10: Favourite (pre-defined) search queries.

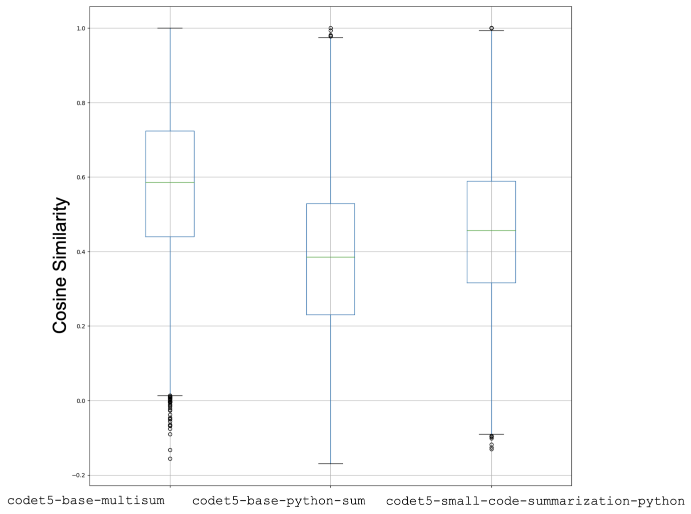
VII. Evaluation

This section presents the evaluations performed to RepoGraph against several criteria, such as function summarization performance, the correctness and relevance of search results, or the processing time for constructing graphs of various sizes.

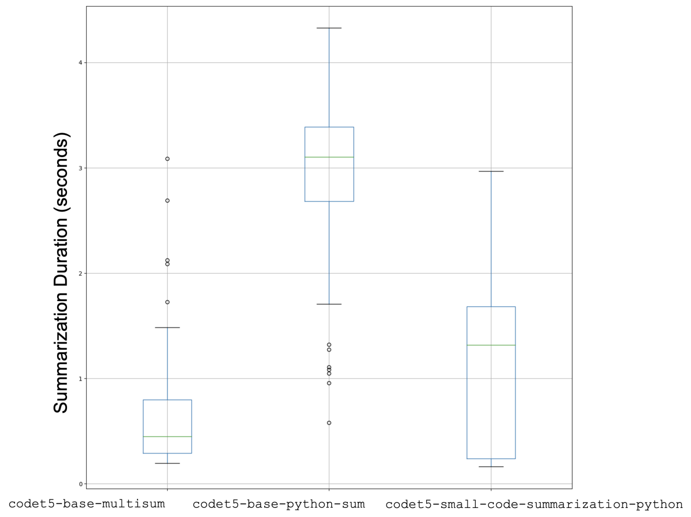
A. Function Summarization

In this work, we have evaluated the summarization models introduced in Section V-D (codet5-base-multi-sum, codet5-small-code-summarization-python, codet5-base-python-sum), using CodeXGLUE Python code snippets [4] as the test dataset. To assess the quality of their summarizations, the associated docstrings in the test dataset were compared to the generated summarizations by applying multi-qa-distilbert-cos-v1 model to generate their embeddings. Note that this model is also employed in the *Search Service* (see Section V-E1), for providing the semantic code search mechanism. Later the cosine similarity between each pair embeddings (associated docstrings and generated summary) was calculated obtaining at the end the similarity score by each model for this dataset.

Figure 11a shows that codet5-base-multi-sum outperforms the other models, codet5-base-python-sum



(a) Function summarization similarity scores.



(b) Function summarization duration.

Fig. 11: Function summarization evalutions.

and `codet5-small-code-summarization-python`, with an average cosine similarity of 0.577. Furthermore, we also evaluated the time needed for each of those models to perform the summarizations of the test dataset. As Figure 11b shows, `codet5-base-multi-sum` outperforms the other models, with most observed execution times below 1 second. Therefore, `codet5-base-multi-sum` was selected as the model to use in the *Summarization Service*

B. Correctness

To assess the correctness of the knowledge graphs generated by `RepoGraph`, several manual queries were devised to compare the generated knowledge graphs against the reported output from `inspect4py`. Table V lists the entities extracted as reported by the output from `inspect4py` using six open-source repositories. This is compared with Table VI, which lists the corresponding entities reported in the subsequently generated knowledge graph.

Tables V and VI show that the number of extracted functions and classes are the same. The number of extracted files for all repositories in Table VI is less than the corresponding entry in Table V. This is because `inspect4py` includes non-Python files, while `RepoGraph` ignore them. This also affects to the number of folders, since `inspect4py` includes folders with non-python files, and those are ignored in `RepoGraph`.

Repository	Folders (inc. root)	Files	Classes	Functions
pyLODE	5	21	4	37
black	35	223	108	367
flake8	15	103	42	328
fastapi	155	2064	551	2246
py2neo	24	126	138	550
pygorithm	16	129	73	246

TABLE V: Entities from `inspect4py` output

Repository	Folders (inc. root)	Files	Classes	Functions
pyLODE	5	20	4	37
black	35	187	108	367
flake8	15	72	42	328
fastapi	157	1130	551	2246
py2neo	24	89	138	550
pygorithm	18	118	73	246

TABLE VI: Entities from `RepoGraph` knowledge graphs

C. Performance

We have also evaluated the performance of `RepoGraph` using repositories of various sizes, in terms of number of entity nodes and relationships of their resulting knowledge graphs. We measured the total time (processing time) to create knowledge graphs for 100 repositories [15]. In this experiment, function summarizations were disabled.

Figure 12a shows the relationship between the number of nodes created and processing time to create those. While Figure 12b plots the number of relationships created against their processing time. Note that each dot represents repository, and consequently a knowledge graph.

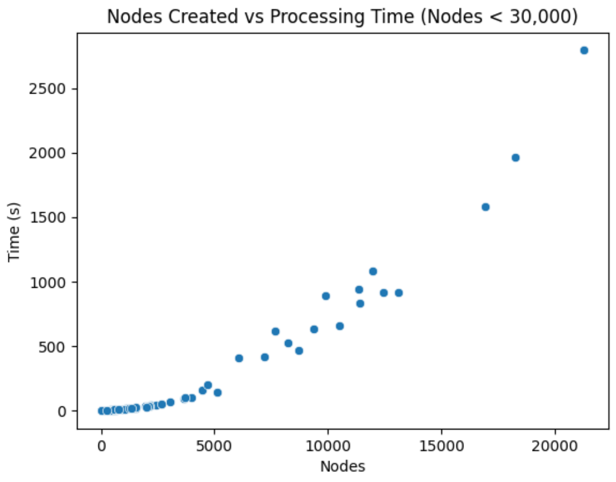
In Figure 12 (a,b) several results were filtered out to remove outliers at the upper end of upload durations that make the graph difficult to understand (we selected repositories with less than 30,000 nodes/ relationships). The the original graphs (without filtering repositories) are available at [16]

In both plots from Figure 12, we observe a roughly exponential relationship with most repositories completing in under 25 minutes (1500 seconds) and a significant number completing in less than 9 minutes (approx. 500 seconds).

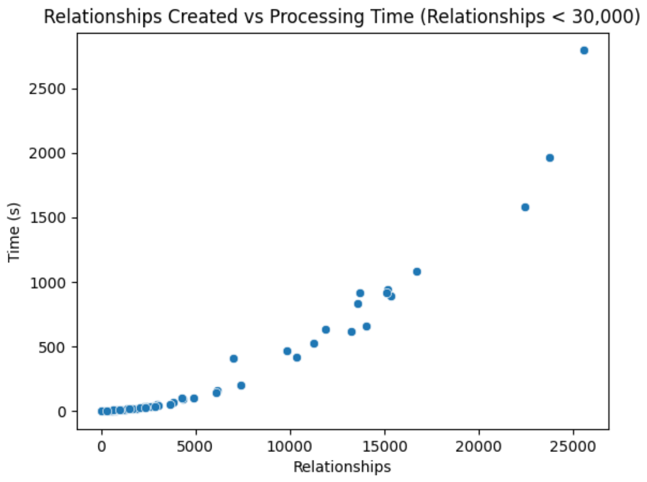
A second experiment was performed later to build knowledge graphs that include function summarizations for a smaller subset (six open-source repositories). The results for these repositories, with and without function summarizations, are listed in Tables VII and VIII, respectively.

[15]Those repositories have been also used to evaluate `inspect4py`: https://github.com/SoftwareUnderstanding/inspect4py/blob/main/evaluation/software_type/software_type_benchmark.csv

[16]https://drive.google.com/file/d/1wdzwQvBmmhns2wqCumDdDe-FnD7yMHyi/view?usp=sharing



(a) Entity Nodes created vs Processing Time



(b) Relationship created vs Processing Time

Fig. 12: Repository processing time evaluations. A dot represent a repository's knowledge graph.

Repository	Time (s)	Entity Nodes	Relationships
pyLODE	7.325	471	741
flake8	45.386	2185	2884
black	55.618	2687	3683
pygorithm	73.6906	3039	3820
py2neo	161.091	4441	6149
fastapi	619.773	7695	13238

TABLE VII: Processing times without summarizations.

We can see an average 4x increase in execution time when summarizations are enabled. For the *fastapi* repository, this leads to an execution time of aprox. 30 minutes. Table VIII also highlights an increase in the number of nodes and relationships created. This is due to `Docstring` nodes (and subsequent `Documents` relationships) for `Function` nodes that do not already have a `Docstring` node.

Repository	Time (s)	Entity Nodes	Relationships
pyLODE	40.600	510	780
flake8	176.130	2302	3001
black	233.173	3045	4044
pygorithm	318.049	3424	4205
py2neo	564.566	5538	7246
fastapi	1759.812	12307	19946

TABLE VIII: Processing times with summarizations.

VIII. Related Work

Several web tools have been developed for enabling users to explore software code. In this section we review those most relevant to our work.

A. *KG4Py*

K4GPy [7] a toolkit for generating Python knowledge graphs and code semantic search, that utilises *LibCST*[17] to perform code analysis on Python files [7] rather than `inspect4py`.*LibCST* uses aspects of Concrete Syntax Trees (CSTs) rather than `inspect4py`'s ASTs to parse Python code, with the additional benefit that CSTs can be used to reconstruct the original source code. `inspect4py` navigates this issue by simply including the original source code in its JSON output and, as such, the net result is broadly similar.

KG4Py also utilises a different ontology because it is beholden to the same feature of property graphs, making utilising existing ontologies infeasible. The *KG4Py* ontology includes six entity types [7, Table 3] and 8 relationship types [7, Table 4], which is significantly less than this work (see Section III).

KG4Py differs from `RepoGraph` in several aspects. With *KG4Py* users can only perform code searchers, while with `RepoGraph`, users not only can do this tasks, as well as visualizing the graph of their uploaded repositories, and explore different features, as well as list their possible issues. With *KG4Py*, users can not store more than one graph, while with `RepoGraph` users are able to store many graphs.

B. *GraphGen4Code*

GraphGen4Code [13] is a 'toolkit for generating code knowledge graphs'. Unlike *KG4py*, it strongly emphasises modelling the data and control flow of Python programs. This is achieved through an extension to the *WALA*analysis[18] library that extracts the control flow of the analysed programs. At the time of publication, the *GraphGen4Code* extension to*WALA* only provides support for Python programs, and, as such, *GraphGen4Code* is only implemented and tested on Python code. The authors note that the 'mechanism can be applied to other programming languages like Javascript' [13].

GraphGen4Code takes a more semantic approach and outputs graphs represented as Resource Description Framework (RDF) triples. This would allow graphs to be stored in a compatible triple store and queried with *SPARQL* [19]. A novel feature of *GraphGen4Code* is that it utilises forum posts from

[17]LibCST: https://github.com/Instagram/LibCST
[18]https://researcher.watson.ibm.com/researcher/view_group.php?id=2999
[19]https://www.w3.org/TR/rdf-sparql-query/

platforms such as *StackOverflow* to augment the knowledge graph. While it does not implement any semantic search functionality, such information could significantly improve the quality of semantic search results versus approaches that rely solely on the knowledge extracted purely from source code.

Our `RepoPyOnto` differs from *GraphGen4Code*, since our ontology represents Python code repositories, rather than just Python software applications. Therefore, `RepoPyOnto` captures features as license, requirement file, README files, etc, which are not available currently in *GraphGen4Code*.

C. Code Smells

Several works [14], [15] utilise knowledge graphs to detect 'bad code smells'. These are anti-patterns and other features that may indicate suboptimal code. Suboptimal code includes poor performance or correctness and code that may be hard to read or maintain. These works had served as an inspiration for a similar feature in `RepoGraph`. Whilst existing works generally evaluate a wider variety of code smells, they provide this functionality in isolation rather than as part of a broader suite of features that leverage a knowledge graph, as we do in `RepoGraph`. Furthermore, our work focused on issues at the 'repository level', rather than at the 'code level'.

IX. Conclusions and Future Work

In this paper, we have presented `RepoGraph`, a new semantic code exploration tool for Python repositories. It facilitates developers in analyzing and extracting knowledge from Python repositories, conducting semantic code searches, visualizing main features, and identifying potential issues. We have introduced a new ontology, `RepoPyOnto`, to represent Python code repositories using features extracted from inspect4py and inferred by `RepoGraph` itself. Our approach combines knowledge graphs, deep transfer learning, and Semantic Web techniques to formalize and connect insights derived from the analysis of Python repositories.

This research demonstrates how deep learning models and knowledge graphs can revolutionize the interaction between developers (from both academia and industry) and Python code repositories, enabling software teams to construct and maintain mental models of their software's operation. Our future plans include enabling `RepoGraph` to generate graphs through syncing with Git repositories, automatically updating knowledge graphs when relevant branches in mirrored Git repositories undergo changes. We also aim to enhance the 'Repository Issues' feature to detect anti-patterns and code smells. Additionally, we intend to improve both function summarization and semantic code search by experimenting with other NLP language models, such as UniXcoder [16], while enhancing the performance of our `codet5-base-python-sum`.

References

[1] P. Naur, Programming as Theory Building, Microprocessing and ¡icroprogramming 15 (5) (1985) 253–261.

[2] V. Lonati, A. Brodnik, T. Bell, A. P. Csizmadia, L. De Mol, H. Hickman, T. Keane, C. Mirolo, M. Monga, What we talk about when we talk about programs, in: Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education, ITiCSE-WGR '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 117–164. `doi:10.1145/3571785.3574125`.
URL https://doi.org/10.1145/3571785.3574125

[3] B. Abu-Salih, Domain-specific knowledge graphs: A survey, Journal of Network and Computer Applications 184 (2021) 103076. `doi:10.1016/j.jnca.2021.103076`.
URL https://doi.org/10.1016/j.jnca.2021.103076

[4] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, S. Liu, CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation, CoRR abs/2102.04664 (2021).

[5] S. Roy, A. Modak, D. Barik, S. Goon, An overview of semantic search engines, Int. J. Res. Rev 6 (10) (2019) 73–85.

[6] L. D. Grazia, M. Pradel, Code search: A survey of techniques for finding code, ACM Computing Surveys 55 (11) (2023) 1–31. `doi:10.1145/3565971`.
URL https://doi.org/10.1145%2F3565971

[7] L. Liang, Y. Li, M. Wen, Y. Liu, KG4Py: A toolkit for generating Python knowledge graph and code semantic search, Connection Science 34 (1) (2022) 1384–1400. `arXiv:https://doi.org/10.1080/09540091.2022.2072471`, `doi:10.1080/09540091.2022.2072471`.

[8] R. Filgueira, D. Garijo, Inspect4py: A Knowledge Extraction Framework for Python Code Repositories, in: IEEE/ACM 19th International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022, IEEE, 2022, pp. 232–236. `doi:10.1145/3524842.3528497`.
URL https://dgarijo.com/papers/inspect4py_MSR2022.pdf

[9] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, A. Taylor, Cypher: An evolving query language for property graphs, in: Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 1433–1445. `doi:10.1145/3183713.3190657`.

[10] Y. Wang, W. Wang, S. Joty, S. C. H. Hoi, CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation (2021). `doi:10.48550/ARXIV.2109.00859`.
URL https://arxiv.org/abs/2109.00859

[11] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, M. Brockschmidt, CodeSearchNet Challenge: Evaluating the State of Semantic Code Search (2019). `doi:10.48550/ARXIV.1909.09436`.
URL https://arxiv.org/abs/1909.09436

[12] N. Reimers, I. Gurevych, Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks (2019). `doi:10.48550/ARXIV.1908.10084`.

[13] I. Abdelaziz, J. Dolby, J. McCusker, K. Srinivas, A Toolkit for Generating Code Knowledge Graphs, in: Proceedings of the 11th on knowledge capture conference, 2021, pp. 137–144.

[14] I. C. Betancourt, N. S. Martínez, M. N. García, O. R. Grass, O. C. Vázquez, Exploiting an Ontology-Based Solution to Study Code Smells, in: Futuristic Trends in Network and Communication Technologies: Third International Conference, FTNCT 2020, Taganrog, Russia, October 14–16, 2020, Revised Selected Papers, Part I 3, Springer, 2021, pp. 234–246.

[15] Y.-p. Cheng, J.-R. Liao, An Ontology-based Taxonomy of Bad Code Smells, in: Proceedings of the Third Conference on IASTED International Conference: Advances in Computer Science and Technology, Phuket, Thailand, 2007.

[16] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, J. Yin, Unixcoder: Unified cross-modal pre-training for code representation, in: S. Muresan, P. Nakov, A. Villavicencio (Eds.), Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022, Association for Computational Linguistics, 2022, pp. 7212–7225. `doi:10.18653/v1/2022.acl-long.499`.
URL https://doi.org/10.18653/v1/2022.acl-long.499