GRAPH4CODE: A Machine Interpretable Knowledge Graph for Code

Ibrahim Abdelaziz [a,*], Julian Dolby [a] James P. McCusker [b] and Kavitha Srinivas [a]
[a] *T.J. Watson Research Center, IBM Research, NY, USA*
E-mails: ibrahim.abdelaziz1@ibm.com, dolby@us.ibm.com, kavitha.srinivas@ibm.com
[b] *Rensselaer Polytechnic Institute (RPI), NY, USA*
E-mail: mccusj2@rpi.edu

Abstract. Knowledge graphs have proven extremely useful in powering diverse applications in semantic search and natural language understanding. GRAPH4CODE is a knowledge graph about program code that can similarly power diverse applications such as program search, code understanding, refactoring, bug detection, and code automation. The graph uses generic techniques to capture the semantics of Python code: the key nodes in the graph are classes, functions and methods in popular Python modules. Edges indicate *function usage* (e.g., how data flows through function calls, as derived from program analysis of real code), and *documentation* about functions (e.g., code documentation, usage documentation, or forum discussions such as StackOverflow). We make extensive use of named graphs in RDF to make the knowledge graph extensible by the community. We describe a set of generic extraction techniques that we applied to over 1.3M Python files drawn from GitHub, over 2,300 Python modules, as well as 47M forum posts to generate a graph with over 2 billion triples. We also provide a number of initial use cases of the knowledge graph in code assistance, enforcing best practices, debugging and type inference. The graph and all its artifacts are available to the community for use.

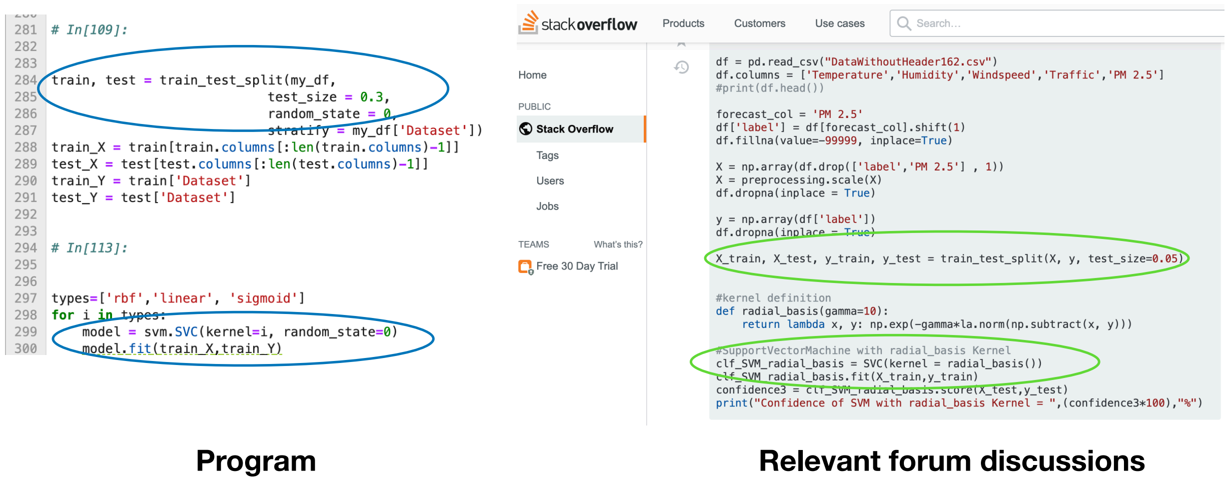
Keywords: Knowledge Graph, Code Analysis, Code Ontology

1. Introduction

Knowledge graphs (e.g. DBpedia [1], Wikidata [2], Freebase [3], YAGO [4], NELL [5]) provide advantages for applications such as semantic parsing [6], recommendation systems [7], information retrieval [8], question answering [9, 10] and image classification [11]. Inspired by such knowledge graphs, we build GRAPH4CODE, a knowledge graph for program code. Many applications could benefit from such knowledge graphs, e.g. code search, code automation, refactoring, bug detection, and code optimization [12], and there many open repositories of code for material. In 2019-2020 alone, there have been over 100 papers[1] using machine learning for problems that involve code, including problems that span natural language and code (e.g., summarizing code in natural language [13]). A knowledge graph that represents code along with natural language descriptions could enhance this research.

We illustrate the value of such a knowledge graph with Figure 1, which shows an example of code search: a developer searches for StackOverflow posts relevant to the Python code from GitHub in the left panel. That code uses `sklearn` to split data into train and test sets (`train_test_split`), and creates an SVC model (`svm.SVC`) to train on the dataset (`model.fit`). On the right is a real post from StackOverflow relevant to this code, in that the code performs similar operations. However, treating program code as text or as an Abstract Syntax Tree (AST) makes this similarity extremely hard to detect. For instance, there is no easy way to tell that the `model.fit` is a call to the same library as `clf_SVM_radial_basis.fit`. Analysis must reveal that `model` and `clf_SVM_radial_basis` refer to the same type of object, and that the result of

[*] All authors contributed equally.
[1] https://ml4code.github.io/papers.html



```
# In[109]:


train, test = train_test_split(my_df,
                               test_size = 0.3,
                               random_state = 0,
                               stratify = my_df['Dataset'])
train_X = train[train.columns[:len(train.columns)-1]]
test_X = test[test.columns[:len(test.columns)-1]]
train_Y = train['Dataset']
test_Y = test['Dataset']


# In[113]:


types=['rbf','linear', 'sigmoid']
for i in types:
    model = svm.SVC(kernel=i, random_state=0)
    model.fit(train_X,train_Y)
```

```
df = pd.read_csv("DataWithoutHeader162.csv")
df.columns = ['Temperature','Humidity','Windspeed','Traffic','PM 2.5']
#print(df.head())

forecast_col = 'PM 2.5'
df['label'] = df[forecast_col].shift(1)
df.fillna(value=-99999, inplace=True)

X = np.array(df.drop(['label','PM 2.5'] , 1))
X = preprocessing.scale(X)
df.dropna(inplace = True)

y = np.array(df['label'])
df.dropna(inplace = True)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.05)


#kernel definition
def radial_basis(gamma=10):
    return lambda x, y: np.exp(-gamma*la.norm(np.subtract(x, y)))

#SupportVectorMachine with radial_basis Kernel
clf_SVM_radial_basis = SVC(kernel = radial_basis())
clf_SVM_radial_basis.fit(X_train,y_train)
confidence3 = clf_SVM_radial_basis.score(X_test,y_test)
print("Confidence of SVM with radial_basis Kernel = ",(confidence3*100),"%")
```

Fig. 1. Code search example

`sklearn.train_test_split` is an argument to `fit` in both programs. Such an abstract representation would ease understanding semantic similarity between programs, making searches for relevant forum posts more precise. Yet most representations in the literature have been application-specific, and most have relied on surface forms such as tokens or ASTs.

We present generic techniques for building representations of code that enable applications such as in Figure 1. We deploy state of the art program analysis that generalizes across programming languages, and we show scaling to millions of programs. From public code, we extract a graph per program, which captures that program in terms of data and control flow. As an example, the program in Figure 1 has a data flow edge from a node denoting the `sklearn.train_test_split` to a node denoting `sklearn.svm.SVC.fit`. It would also have a control flow edge from `sklearn.svm.SVC` to `sklearn.svm.SVC.fit`. Representing programs as data flow and control flow is crucial because programs that behave similarly can look arbitrarily different at a token or AST level due to syntactic structure or choices of variable names. Conversely, programs that look similar (e.g., they invoke a call to a method called `fit`) can have entirely different meanings. We build such representations for 1.3 million Python programs on GitHub, each analyzed into its own separate graph.

While these graphs capture how libraries get used, they are not sufficient; there are semantics in textual documentation of libraries, and in forum discussions of them. We therefore link library calls to documentation and forum discussions. We identified commonly used modules in Python and their dependencies, and added canonical nodes to represent each class, function or method for 2300+ modules. These nodes are linked using information retrieval techniques to the documentation of method and classes (from usage documentation, when available, or from direct introspection), and to forum posts which specify the method or the class (an example of which is shown in Figure 1 for the `sklearn.svm.SVC.fit` method). We performed this linking for 257K classes, 5.8M methods, and 278K functions, and processed 47M posts from StackOverflow and StackExchange. To our knowledge, this is the first knowledge graph that captures the semantics of code, and it associated artifacts.

Our knowledge graph is an RDF graph since RDF provides good abstractions for modeling individual program data such as named graphs, and SPARQL, the query language for RDF provides extensive support for the sorts of operations that are crucial for understanding control and data flow in programs, especially transitivity. Each program graph is modeled separately, and we use existing properties from ontologies such as schema.org, SKOS, DublinCore, and SemanticScience Integrated ontology (SIO) to model relationships between program and documentation entities.

In summary, our key contributions are as follows:

– A comprehensive knowledge graph for 1.3 million Python programs on GitHub
– A model to represent code and its natural language artifacts (Section 2)

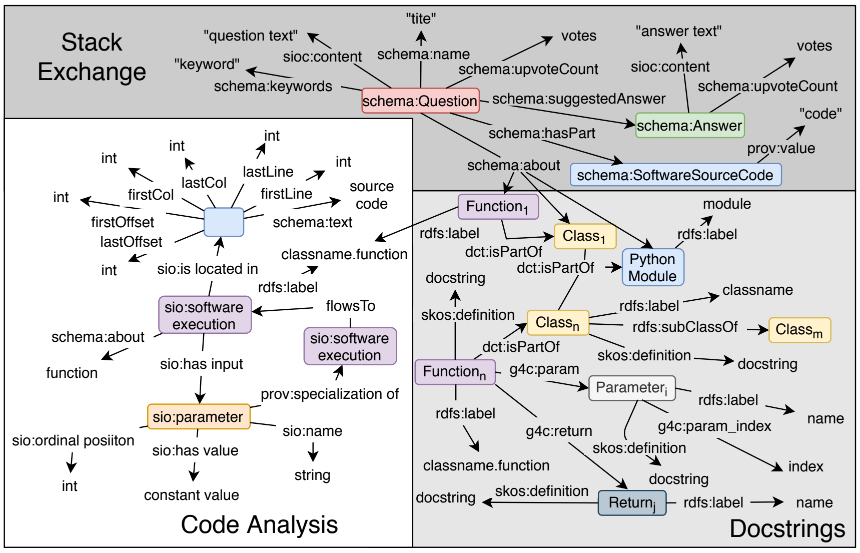


Fig. 2. A concept map of GRAPH4CODE's overall schema, across the code analysis, Stack Exchange, and Docstrings extractions.

- A language neutral approach to mine common code patterns (Section 3)
- Connections from code to associated forum discussions and documentation (Section 4)
- Use cases showing generality and promise of GRAPH4CODE (Section 5)

All artifacts associated with GRAPH4CODE, along with detailed descriptions of the modeling, use cases, query templates and sample queries for use cases are available at https://wala.github.io/graph4code/.

2. Ontology and Modeling

Figure 2 shows GRAPH4CODE's graph model, which is based on the Semanticscience Integrated Ontology (SIO) [14], and Schema.org classes and properties[2]. Classes and functions in code are modeled as `g4c:Class` and `g4c:Function`[3], and each has a URI based on its python import path along with the `python:` prefix[4]. For instance, the function `pandas.read_csv` in Figure 1 is `python:pandas.read_csv`. Each invocation of a function is an instance of `sio:SoftwareExecution`.

[2]The full schema specified as TTL files can be found at https://github.com/wala/graph4code/tree/master/docs

[3]`g4c` is http://purl.org/twc/graph4code/ontology/

[4]http://purl.org/twc/graph4code/python/

Function invocations in code analysis link to their functions by RDFS label. However, since the return types of functions are often unknown in Python, this linkage is not always predictable. For instance, in the right panel of Figure 1, the call `df.fillna` will reflect the analysis to that point, its RDFS label would be `pandas.read_csv.fillna`. Modeling of code analysis is detailed in Section 3.

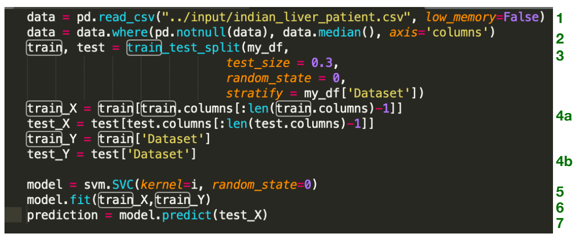
We model StackExchange questions and answers using properties and classes from Schema.org, while expressing the actual question text as `sioc:content`. We chose Schema.org because it models the social curation of questions and answers across the web. Mentions of specific Python classes, modules, and functions in the forum posts are linked to classes and function URIs using `schema:about`. We also extract software snippets in forum posts, and use `schema:SoftwareSourceCode` to express source code snippets from questions and answers.

The resulting knowledge graph allows the querying of usage patterns for python functions and classes directly by URI, along with any forum post or documentation associated with them (See Section 5).

3. Mining Code Patterns

3.1. Extraction of Python Files from GitHub

Our starting point to build GRAPH4CODE is 1.38 million code files from GitHub. To extract this dataset,



```
data = pd.read_csv("../input/indian_liver_patient.csv", low_memory=False)   1
data = data.where(pd.notnull(data), data.median(), axis='columns')          2
train, test = train_test_split(my_df,                                       3
                               test_size = 0.3,
                               random_state = 0,
                               stratify = my_df['Dataset'])
train_X = train[train.columns[:len(train.columns)-1]]                       4a
test_X = test[test.columns[:len(test.columns)-1]]
train_Y = train['Dataset']
test_Y = test['Dataset']                                                    4b

model = svm.SVC(kernel=i, random_state=0)                                   5
model.fit(train_X,train_Y)                                                  6
prediction = model.predict(test_X)                                          7
```

Fig. 3. Illustrative code example from GitHub

we ran a SQL query on the Google BigQuery dataset[5]. The query looks for all Python files and Python notebooks from repositories that had at least two watch events associated with it, and excludes large files. Duplicate files were eliminated, where a duplicate was defined as having the same MD5 hash as another file in the dataset.

3.2. *Code Analysis*

Figure 3 extends the code in our running example (Figure 1) to illustrate how we construct the knowledge graph. In this example, a CSV file is first read using the Pandas library with a call to `pandas.read_csv`, with the call being represented as `1` on the right of the program. The object returned by the call has an unknown type, since most code in Python is not typed. Some filtering is performed on this object by filling in the missing values with a median with a call to `where`, which is call `2`. The object returned by `2` is then split into train and test with a call to `train_test_split`, which is call `3`. Two subsets of the train data are created (`4`) which are then used as arguments to the `fit` call (`6`), after creating the `SVC` object in call (`5`). The test data is similarly split into its X and Y components and used as arguments to the `predict` call (`7`).

Figure 4 illustrates the output of program analysis. For this, we use WALA[6] – an open source library which can perform inter-procedural program analysis for Python, Javascript, and Java. The output of WALA is a control flow (depicted as green edges) and data flow (depicted as blue edges) graph for each analyzed program. The control flow edges are straightforward in this example and capture the order of calls, and this is particularly useful when data flow need not be explicit,

[5]https://github.com/wala/graph4code/blob/master/extraction_queries/bigquery.sql

[6]https://github.com/wala/WALA

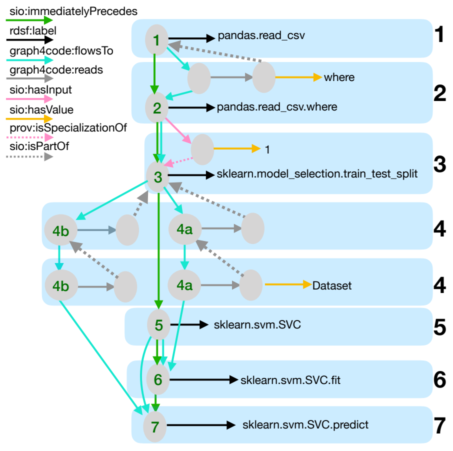


Fig. 4. Dataflow graph for the running example

such as when a `fit` call (labeled 6) must precede a `predict` call (labeled 7) for the `sklearn` library.

We discuss the data flow shown in Figure 4 in more detail to illustrate assumptions in our analysis and to describe our modeling. This figure is a subset of the actual model, but we show all the key relations at least once. This graph shows two key relations that capture the flow through the code:

flowsTo [7] (blue edges) captures the notion of data flow from one node to another, abstracting away details like argument numbers or names. Many application queries can be expressed as transitive paths on `flowsTo`. We use SPARQL's property path operators extensively to accomplish this.

immediatelyPrecedes (green edges) captures code order. Queries such as `predict` calls not preceded by `fit` calls can be expressed this way.

Node `1` in Figure 4 corresponds to the execution of `read_csv`. Since we are trying to understand mainly how code uses libraries rather than the libraries themselves, we assume any call simply returns a new object, with no side effects[8]. In Figure 3, we see `data` used as the receiver to the `where` (i.e. the object on which the call is made). Since Python does not have methods per se[9], this call has two steps as shown in Figure 4:

[7]predicates use the `graph4code:` prefix, http://purl.org/twc/graph4code/

[8]Of course this may not be true, but allows us to scale analysis without delving into the analysis of libraries, which may be huge and, in languages like Python, are often written in C.

[9]It is legal to modify "methods", e.g. `data.where = lambda...`.

1. A read of the `where` property from `data`. It is a blank node with a `read` edge to another blank node. The read node has a `hasValue` edge to the name of the field `where` and an `isPartOf` edge to the object being read.
2. A call `B`, which is given the label `pandas.read_csv.where` since it represents the actual invocation of `where` on the object returned by the `pandas.read_csv` call.

We see from Figure 3 that `data` is used as argument 1 to `train_test_split`, labelled `3`. The graph denotes argument numbers using blank nodes as shown in Figure 4: there is a `hasInput` edge to the blank node which, in turn, has a `hasValue` to the argument number 1 and an `isSpecializationOf` edge to the call. All arguments have this representation, but we show only this one to minimize clutter.

The call to `test_train_split` returns a tuple, which is split into `train` and `test`. This is captured in the first box labeled `4`. Then each of `train` and `test` are split into their X and Y components for learning, which is shown in the second box label `4`. The `train` components is used for `fit`, and the `4a` node show the read of the test components from `Dataset`. The `4b` nodes follow the test component to the `predict` call; the X field is a slice and so does not have a specific field. Note that this example does not have dataflow directly from `fit` to `predict`, but this graph also captures the ordering constraint between them as shown in Figure 4.

Each program is analyzed into a separate RDF graph, with connections between different programs being brought about by `rdfs:label` to common calls (e.g., `sklearn.svm.SVC.fit`). Note that this type of modeling accommodates the addition of more graphs easily as more code gets available for analysis.

4. Linking Code to Documentation and Forum Posts

4.1. Extracting Documentation into the Graph

To generate documentation for all functions and classes used in the 1.3 million files, we first processed all the import statements to gather popular libraries (libraries with more than 1000 imports across the files). 506 such libraries were identified, of which many were included in the Python language themselves. For each of these libraries, we tried to resolve their location on GitHub to get not only the documentation embedded in its code, but also associated usage documentation which tends to be in the form of markdown or restructured text files. We found 403 repositories using web searches on GitHub for the names of modules.

The first source of documentation we collected is from the documentation embedded in these code files. However, documentation in the source is insufficient because Python libraries often depend on code written in other languages. As an example, the `tensorflow.placeholder` function is defined in C, and has no stub that allows the extraction of its documentation. Therefore, to gather additional documentation for the popular libraries, we created a virtual environment, installed it, and used Python `inspect` to gather the documentation. Currently, we only gather information about the latest version of the code. This is currently a limitation of the graph that we will address in future work.

This step yielded 6.3M pieces of documentation for functions, classes and methods in 2300+ modules (introspection of each module brought in its dependencies). The extracted documentation is added to our knowledge graph where, for each class or function, we store its documentation string, base classes, parameter names and types, return types and so on. As an example, the following are some materialized information about `pandas.read_csv` in GRAPH4CODE:

```
py:pandas.read_csv
    a                   graph4code:Function;
    dcterms:isPartOf    py:pandas;
    skos:definition
        "Read a comma-separated values ...";
    g4c:return          py:pandas.read_csv/r/1 ;
    g4c:return_inferred_type
                        py:pandas.DataFrame ;
    g4c:param           py:pandas.read_csv/p/1 ;
    g4c:param           py:pandas.read_csv/p/2 .

py:pandas.read_csv/r/1
    a                 graph4code:Return ;
    g4c:return_index  1 ;
    skos:definition   "A comma-separated values ..." .

 py:pandas.read_csv/p/2
    a                   graph4code:Parameter ;
    rdfs:label          "sep" ;
    g4c:param_index     2 ;
    g4c:param_inferred_type py:str ;
    skos:definition    "Delimiter to use ... " .
```

4.2. Extraction of StackOverflow and StackExchange Posts

User forums such as Stackoverflow[10] provide a lot of information in the form of questions and answers about code. Moreover, user votes on questions and answers can indicate the value of the question and the correctness of the answer. While Stackoverflow is geared towards programming, StackExchange[11] provides a network of sites around many other topics such as data science, statistics, mathematics, and artificial intelligence.

To further enrich our knowledge graph with natural language posts about code and other documentation, we linked each function, class and method to its relevant posts in Stackoverflow and StackExchange. In particular, we extracted 45M posts (question and answers) from StackOverflow and 2.7M posts from StackExchange in Statistical Analysis, Artificial Intelligence, Mathematics and Data Science forums. Each question is linked to all its answers and to all its metadata information like tags, votes, comments, codes, etc.

We then built an elastic search index for each source, where each document is a single question with all its answers. The documents were indexed using a custom analyzer tailored for natural language as well as code snippets. Then, for each function, class and method, we perform a multi-match search[12] over this index to retrieve the most relevant posts (a limit of 5K matches per query is imposed) and link it to the corresponding node in the knowledge graph. For example, the following question is linked to `SVC` class:

```
<https://stackoverflow.com/questions/47663694>
    rdf:type        schema:Question;
    schema:about    py:sklearn.svm.SVC;
    schema:name
        "How to run SVC classifier..";
    schema:suggestedAnswer
    <https://stackoverflow.com/questions/a/47664483>.

<https://stackoverflow.com/questions/a/47664483>
    rdf:type        schema:Answer;
    sioc:content
    "Build your classifier: classifier = svm.SVC...".
```

4.3. Extracting Class Hierarchies

As in the case of extracting documentation embedded in code, extraction of class hierarchies was based on Python introspection of the 2300+ modules. For example, the below triples list some of the subclasses of `BaseSVC`:

```
py:sklearn.svm.SVC rdfs:subClassOf
                    py:sklearn.svm._base.BaseSVC .
py:sklearn.svm.NuSVC rdfs:subClassOf
                    py:sklearn.svm._base.BaseSVC .
```

5. Knowledge Graph: Properties and Uses

As discussed earlier, a large literature exists on using various code abstractions such as ASTs, text, or even output of program analysis artifacts for all sorts of applications such as code refactoring, code search, code de-duplication detection, debugging, enforcement of best practices etc. The popularity of WALA[13], the program analysis tool used to generate the analysis based representation of GRAPH4CODE, attests to the fact that numerous applications exist for this type of representation of code. To our knowledge, GRAPH4CODE is the first attempt to build a knowledge graph over a large repository of programs and systematically link it to documentation and forum posts related to code. We believe that by doing so, we will enable a new class of applications that combine code semantics as expressed by program flow along with natural language descriptions of code.

5.1. Graph Statistics

Table 1 shows the number of unique methods, classes, and functions in docstrings (embedded documentation in code). These correspond to all documentation pieces we found embedded in the code files or obtained through introspection (see Section 4.1). Overall, we extracted documentation for 278K functions, 257K classes and 5.8M methods. Table 1 also shows the number of links made from other sources to docstrings documentation. Static analysis of the 1.3M code files created a total of 7.3M links (4.2M functions, 2.1M class and 959K methods). We also created links to web forums in Stackoverflow and StackExchange: GRAPH4CODE has currently 106K, 88K and 742K links from web forums to functions, classes and methods, respectively. This results in a knowledge graph with a total of 2.09 billion edges; 75M triples from docstrings, 246M from web forums and 1.77 billion from static analysis.

[10]https://stackoverflow.com/
[11]https://data.stackexchange.com/
[12]https://github.com/wala/graph4code/blob/master/extraction_queries/elastic_search.q
[13]https://github.com/wala/WALA

	Functions(K)	Classes(K)	Methods(K)
Docstrings	278	257	5,809
Web Forums Links	106	88	742
Static Analysis Links	4,230	2,132	959

Table 1

Number of functions, classes and methods in docstrings and the links connected to them from user forums and static analysis. This results in a knowledge graph with a 2.09 billion edges in total

5.2. *Querying* GRAPH4CODE

This section shows basic queries for retrieving information from GRAPH4CODE.

The first query returns the documentation of a class or function, in this case `pandas.read_csv`. It also returns parameter and return types, when known. One can expand these parameters (`?param`) further to get their labels, documentation, inferred types, and check if they are optional.

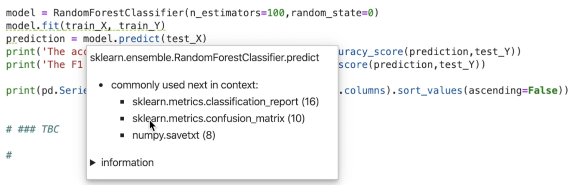
```
select ?doc ?param ?return where {
graph <http://purl.org/twc/graph4code/docstrings>
    {
      ?s  rdfs:label            "pandas.read_csv" ;
          skos:definition       ?doc .
      optional { ?s g4c:param   ?param . }
      optional { ?s g4c:return  ?return . }
    }
}
```

In addition to the documentation of `pandas.read_csv`, we can also get the forum posts that mention this function by appending the following to the query above. This will return all questions in StackOverflow and StackExchange forums about `pandas.read_csv` along with its answers.

```
  graph ?g2 {
        ?ques   schema:about            ?s ;
                schema:name             ?q_title ;
                schema:suggestedAnswer  ?a .
        ?a sioc:content ?answer.
    }
```

Another use of GRAPH4CODE is to understand how people use functions such as `pandas.read_csv`. In particular, the query below shows when `pandas.read_csv` is used, what are the `fit` functions that are typically applied on its output.

```
select distinct ?label where {
  graph ?g {
    ?read   rdfs:label          "pandas.read_csv" .
    ?fit    schema:about        "fit" .
    ?read   graph4code:flowsTo+ ?fit .
    ?fit    rdfs:label          ?label .
  }
}
```



```
model = RandomForestClassifier(n_estimators=100,random_state=0)
model.fit(train_X, train_Y)
prediction = model.predict(test_X)
print('The acc                                    uracy_score(prediction,test_Y))
print('The F1                                     score(prediction,test_Y))

print(pd.Serie                                    .columns).sort_values(ascending=False))


# ### TBC

#
```

sklearn.ensemble.RandomForestClassifier.predict

- commonly used next in context:
  - sklearn.metrics.classification_report (16)
  - sklearn.metrics.confusion_matrix (10)
  - numpy.savetxt (8)

▶ information

Fig. 5. Finding most commonly used next step

5.3. *Uses of* GRAPH4CODE

In addition to simple templates that people can use to query the knowledge graph, we describe three use cases for GRAPH4CODEhere in some detail. Additional use cases can be found in https://wala.github.io/graph4code/.

5.4. *Code Assistance: Next Coding Step*

In this use case, we integrated GRAPH4CODE inside and IDE and used it for code assistance to find the most commonly used next steps, based on the context of their own code. Context, in this query, means data flow predecessors of the node of interest; in this case, we take a simple example of the single predecessor call that constructed the classifier. Figure 5 shows an example of a real Kaggle notebook, where users can select any expression in the code and get a list of the most common next steps along with the frequency with which the next step is observed. For example, in similar contexts after a `model.predict` call, data scientists typically do one of the following: 1) build a text report showing the main classification metrics (frequency: 16), 2) report the confusion matrix which is an important step to understand the classification errors (frequency: 10) and 3) save the prediction array as text (frequency: 8). This can help users by alerting them to best practices from other data scientists. In the example shown above, the suggested step of adding code to compute a confusion matrix is actually useful. The existing Kaggle notebook does not contain this

```
select (count(?g) as ?c) ?next_label where {
    graph ?g {
         ?pred      rdfs:label          "sklearn.ensemble.RandomForestClassifier" .
         ?pred      graph4code:flowsTo+ ?this .
         ?this rdfs:label               "sklearn.ensemble.RandomForestClassifier.predict" .
         ?this rdf:type                 <http://semanticscience.org/resource/SIO_000667> .
         ?this graph4code:flowsTo+      ?next .
         ?next rdfs:label               ?next_label .
         ?next rdf:type                 <http://semanticscience.org/resource/SIO_000667> .
    }
} group by ?next_label order by desc(?c) limit 3
```

Fig. 6. Query to find the next step in a program

call, but the call is very helpful to understand the properties of a classifier.

We show in Figure 6 the SPARQL query used to support this use case. The query variable *?this* is the node that has an execution path of “sklearn.ensemble.RandomForestClassifier.predict“ where the user is at in the program, and has paused to ask next steps from this call. *?pred* refers to predecessors that flow into this node within the program, and it adds context to the call to predict. In this example it is just the constructor - "sklearn.ensemble.RandomForestClassifier". *?next* shows all the calls that tend to be called on the return type of *?this*, ordering by counts across the graphs in the repository. For other code assistance use case, please see [15].

5.4.1. *Enforcing best practices*

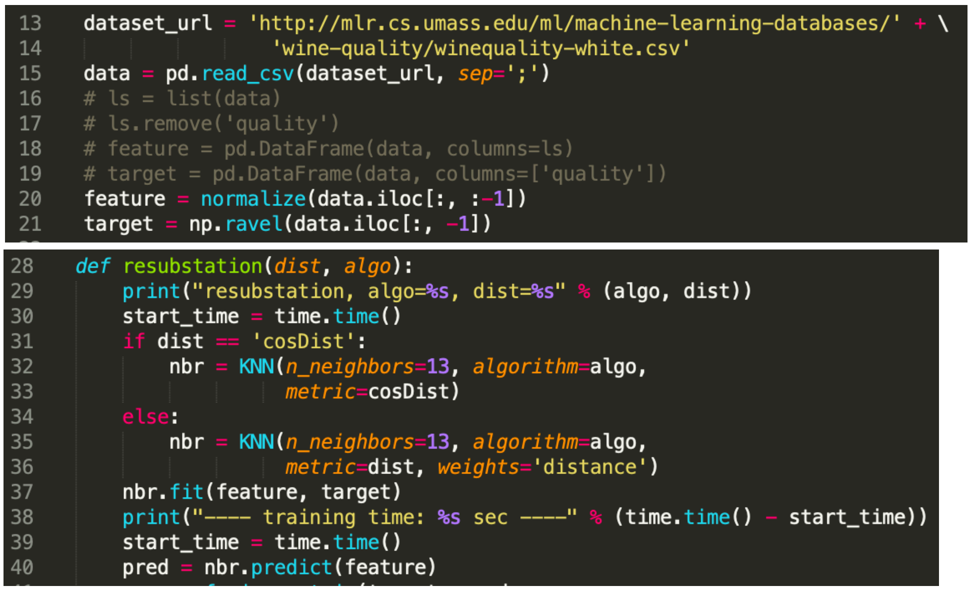
There has been a long history in the programming languages literature of using static analysis to perform the detection of anti patterns or best practices for various application frameworks and systems code (e.g. [16], [17], and [18]). For instance, [18] described a set of anti patterns for usage of enterprise java beans frameworks. Frequently, the detection of anti patterns has been implemented using custom code over the output of static analysis, but see [19] for an approach that uses a custom declarative query language called PQL which can be used to detect anti patterns.

In most cases, detection of several hundred anti-patterns can be enabled with just a handful of query templates ([18]). We enable this use case easily with our knowledge graph. To illustrate how this might work, we describe a best practices pattern for data scientists, which states that any data scientist who builds a machine learning model must use different data to train the model with a `fit` call, than the data they use to ultimately test the validity of the model with a `predict` call. Note that it is perfectly reasonable for the user to also use the `predict` call to assess the goodness of the model on the training data. All that is required is that *some other data* also be used to the predict call. Fortunately, SPARQL 1.1 is expressive enough to describe the anti pattern easily over the knowledge graph. Basically, we look for any dataset that flows into argument 1 of a `fit` call, that also flows into argument 1 of the `predict` call, and filter cases that do have two different datasets flowing into the `predict` calls on the exact same model. We ran this query over the knowledge graph, and found 245 examples of this anti pattern in the graph. Figure 7 shows one such result. Data flows from line 15 to the `fit` call on line 37, and to the `predict` call on line 40 on the same model `nbr`, but no test data is used to validate the model at all.

We stress that this is just one example of the types of templates that can be built to enforce API specific or even organization specific best practices. As noted extensively in the literature, a small set of templates typically are needed to enforce many hundred anti-patterns. For data science scenarios alone, there are two other best practices we have implemented. One checks that users use some type of hyper-parameter optimization techniques when building models rather than setting hyper-parameters to models manually. A second checks that users compares more than one model for each dataset, because it is well known that no single algorithm works best on all datasets. Together these queries act as tutorials for how to construct such anti-pattern detection across a repository of code.

5.4.2. *Debugging with Stackoverflow*

A common use of sites such as StackOverflow is to search for posts related to an issue with a developer's code, often a crash. In this use case, we show an example of searching StackOverflow using the code context in Figure 1, based on the highlighted code locations found with dataflow to the `fit` call. Such a search on GRAPH4CODE does produce the StackOverflow result shown in Figure 1 based on links with the coding context, specifically the `train_test_split` and



```
dataset_url = 'http://mlr.cs.umass.edu/ml/machine-learning-databases/' + \
              'wine-quality/winequality-white.csv'
data = pd.read_csv(dataset_url, sep=';')
# ls = list(data)
# ls.remove('quality')
# feature = pd.DataFrame(data, columns=ls)
# target = pd.DataFrame(data, columns=['quality'])
feature = normalize(data.iloc[:, :-1])
target = np.ravel(data.iloc[:, -1])
```

```
def resubstation(dist, algo):
    print("resubstation, algo=%s, dist=%s" % (algo, dist))
    start_time = time.time()
    if dist == 'cosDist':
        nbr = KNN(n_neighbors=13, algorithm=algo,
                  metric=cosDist)
    else:
        nbr = KNN(n_neighbors=13, algorithm=algo,
                  metric=dist, weights='distance')
    nbr.fit(feature, target)
    print("---- training time: %s sec ----" % (time.time() - start_time))
    start_time = time.time()
    pred = nbr.predict(feature)
```

Fig. 7. An example from detected anti patterns

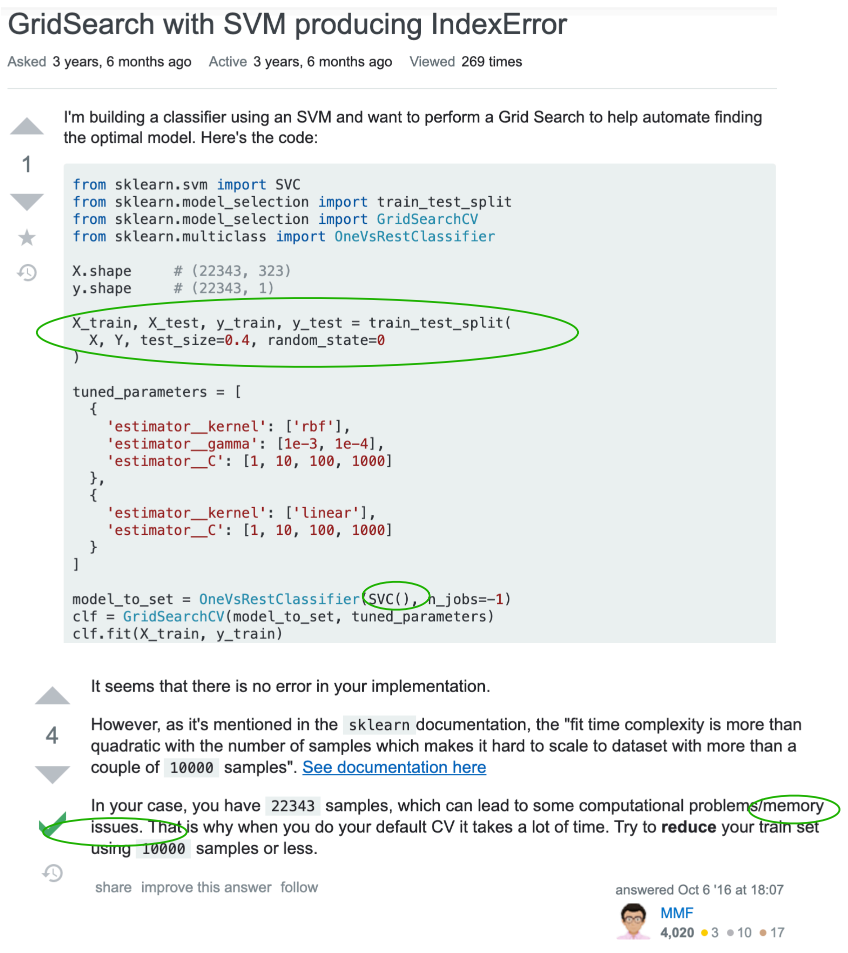
SVC.fit call as one might expect. Suppose we had given SVC a very large dataset, and the fit call had memory issues; we could augment the query to look for posts that mention ‘memory issue’, in addition to taking the code context shown in Figure 1 into consideration. Figure 8 shows the first result returned by such a query over the knowledge graph. As shown in the figure, this hit is ranked highest because it matches both the code context in Figure 1 highlighted with green ellipses, and the terms "memory issue" in the text. What is interesting is that, despite its irrelevant title, the answer is actually a valid one for the problem. A text search on StackOverflow with ‘sklearn’, ‘SVC’ and ‘memory issues’ as terms does not return this answer in the top 10 results. Figure 9 shows the second result, which is the first result returned by a text search on StackOverflow. Note that our system ranks this lower because the coding context does not match the result as closely.

6. Related Work

To our knowledge, there is no comprehensive knowledge graph for code that integrates semantic analysis of code along with textual artifacts about code. Here we review related work around issues of how code has been typically represented in the literature, what sorts of datasets have been available for code, and ontologies or semantic models for code.

6.1. Code Representation

A vast majority of work in the literature has used either tokens or abstract syntax trees as input representations of code (see [12] for a comprehensive survey on the topic). When these input code representations are used for a specific application, the target is usually a distributed representation of code (see again [12] for a breakdown of prior work), with a few that build various types of probabilistic graphical models from code. Few works have used data and control flow based representations of code as input to drive various applications. As an example, [20] used a program dependence graph to detect code duplication on Javascript programs, but the dependence is computed in an intra-procedural manner. Similarly, [21] augments an AST based representation of code along with local data flow and control flow edges to predict variable names or find the misuse of variables in code. [22] combines token based representations of code with edges based on object uses, and AST nodes to predict the documentation of a method. [23, 24] includes partial object use information from WALA for code completion tasks, but the primary abstraction in that work is (a) a vector representation of APIs for Java SWT, that they used in machine learning algorithms such as best matching neighbors to find the next best API for completion [23], or (b) as a Bayesian network which reflects the likelihood of a specific method call given the other method calls that have been observed [24]. [25, 26] employs



GridSearch with SVM producing IndexError

Asked 3 years, 6 months ago Active 3 years, 6 months ago Viewed 269 times

1

I'm building a classifier using an SVM and want to perform a Grid Search to help automate finding the optimal model. Here's the code:

```
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.multiclass import OneVsRestClassifier

X.shape     # (22343, 323)
y.shape     # (22343, 1)

X_train, X_test, y_train, y_test = train_test_split(
  X, Y, test_size=0.4, random_state=0
)

tuned_parameters = [
  {
    'estimator__kernel': ['rbf'],
    'estimator__gamma': [1e-3, 1e-4],
    'estimator__C': [1, 10, 100, 1000]
  },
  {
    'estimator__kernel': ['linear'],
    'estimator__C': [1, 10, 100, 1000]
  }
]

model_to_set = OneVsRestClassifier(SVC(), n_jobs=-1)
clf = GridSearchCV(model_to_set, tuned_parameters)
clf.fit(X_train, y_train)
```

4

It seems that there is no error in your implementation.

However, as it's mentioned in the `sklearn` documentation, the "fit time complexity is more than quadratic with the number of samples which makes it hard to scale to dataset with more than a couple of `10000` samples". See documentation here

In your case, you have `22343` samples, which can lead to some computational problems/memory issues. That is why when you do your default CV it takes a lot of time. Try to **reduce** your train set using `10000` samples or less.

share improve this answer follow

answered Oct 6 '16 at 18:07
MMF
4,020 ●3 ●10 ●17

Fig. 8. First search result for debugging SVC query

a mostly intraprocedural (with heuristics for handling interprocedural analysis) to mine a large number of graphs augmented with control and data flow, for the purposes of code completion for Java API calls. This work is interesting because, like us, [25] it creates a large program graph database which models dependencies between parent and child graphs, from which a Bayesian model is constructed to predict the next set of API calls based on the current one.

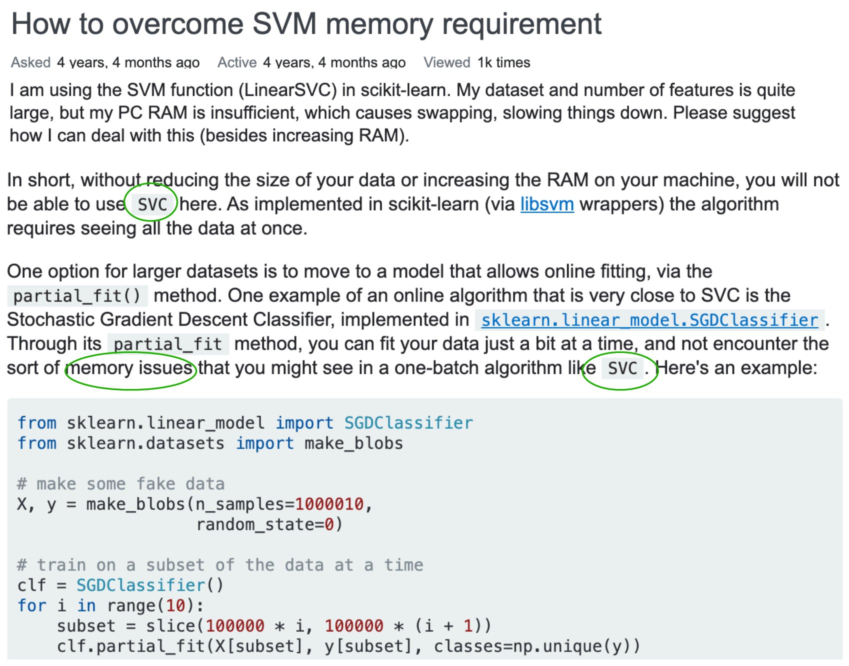
Our work can be distinguished from prior work in this area in two key ways: (a) our work targets interprocedural data and control flow, in the presence of first class functions and no typing, to create a more comprehensive representation of code, and (b) we use this representation to drive the construction of a *multi-purpose* knowledge graph of code that is connected to its textual artifacts.

6.2. Code Datasets

Several research efforts started recently to focus on using machine learning for code summarization [27–29], code search [30] and models such as CodeBERT [31]. The datasets used by these approaches tend to be code- and task-specific. To the best of our knowledge, there is no work that tries to build a general knowledge graph for code and we believe that these approaches can directly benefit from GRAPH4CODE. We, however, leave this for future work.

6.3. Semantic Models of Code

SemanGit [32] is a linked data version based on Github activities. Unlike GRAPH4CODE, SemanGit focus on modeling user activities on Github and not on understanding the code itself as in GRAPH4CODE. CodeOntology [33] in an ontology designed for mod-


How to overcome SVM memory requirement

Asked 4 years, 4 months ago Active 4 years, 4 months ago Viewed 1k times

I am using the SVM function (LinearSVC) in scikit-learn. My dataset and number of features is quite large, but my PC RAM is insufficient, which causes swapping, slowing things down. Please suggest how I can deal with this (besides increasing RAM).

In short, without reducing the size of your data or increasing the RAM on your machine, you will not be able to use `SVC` here. As implemented in scikit-learn (via libsvm wrappers) the algorithm requires seeing all the data at once.

One option for larger datasets is to move to a model that allows online fitting, via the `partial_fit()` method. One example of an online algorithm that is very close to SVC is the Stochastic Gradient Descent Classifier, implemented in `sklearn.linear_model.SGDClassifier`. Through its `partial_fit` method, you can fit your data just a bit at a time, and not encounter the sort of memory issues that you might see in a one-batch algorithm like `SVC`. Here's an example:

```
from sklearn.linear_model import SGDClassifier
from sklearn.datasets import make_blobs

# make some fake data
X, y = make_blobs(n_samples=1000010,
                  random_state=0)

# train on a subset of the data at a time
clf = SGDClassifier()
for i in range(10):
    subset = slice(100000 * i, 100000 * (i + 1))
    clf.partial_fit(X[subset], y[subset], classes=np.unique(y))
```

Fig. 9. Second search result for debugging SVC query

eling code written in Java. The ontology is similar to ours when it comes to modeling relationships among classes and methods. A crucial difference, however, is how the code itself is parsed and hence how it gets modeled. CodeOntology's parser relies on Abstract Syntax Trees (AST) for understanding the semantics of the code while GRAPH4CODE represents programs as data flow and control flow which is crucial because programs that behave similarly can look arbitrarily different at a token or AST level due to syntactic structure or choices of variable names. [34] proposed an approach for learning ontology from Java code using Hidden Markov Models. Unlike this approach, GRAPH4CODE relies on a standard static analysis library (WALA) for code understanding which is then modeled using our ontology proposed earlier. [35] aims to construct knowledge graph of scientific concepts expressed in text books to link it to code artifacts such as function or variable names. In their work, text sentences are converted into triples and linked to source code based on token matches. This is different from GRAPH4CODE's approach where the focus is on modeling control and data flow of code and then linking classes and methods to their documentation. Augmenting GRAPH4CODE with an approach specified in [35] is an interesting idea for future work.

7. Conclusions

We presented Graph4Code, a knowledge graph that connects code analysis with other diverse sources of knowledge about code such as documentation and user-generated content in StackOverflow and StackExchange. To demonstrate the promise of such a knowledge graph, we provided 1) a set of SPARQL templates to help users query the graph, 2) initial use cases in debugging, enforcing best practices and type inference. We hope that this knowledge graph will help the community build better tools for code automation, code search and bug detection and bring semantic web technologies into the programming languages research.

References

[1] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P.N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer and C. Bizer, DBpedia - A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia, *Semantic Web Journal* **6**(2) (2015), 167–195. http://jens-lehmann.org/files/2015/swj_dbpedia.pdf.

[2] D. Vrandečić and M. Krötzsch, Wikidata: A Free Collaborative Knowledgebase, *Commun. ACM* **57**(10) (2014), 78–85. doi:10.1145/2629489.

[3] K. Bollacker, C. Evans, P. Paritosh, T. Sturge and J. Taylor, Freebase: a collaboratively created graph database for struc-

turing human knowledge, in: *In SIGMOD Conference*, 2008, pp. 1247–1250.

[4] F.M. Suchanek, G. Kasneci and G. Weikum, Yago: A Core of Semantic Knowledge, in: *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, ACM, New York, NY, USA, 2007, pp. 697–706. ISBN 978-1-59593-654-7. doi:10.1145/1242572.1242667.

[5] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E.R. Hruschka Jr. and T.M. Mitchell, Toward an Architecture for Never-ending Language Learning, in: *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI'10, AAAI Press, 2010, pp. 1306–1313. http://dl.acm.org/citation.cfm?id=2898607.2898816.

[6] L. Heck, D. Hakkani-Tür and G. Tur, Leveraging knowledge graphs for web-scale unsupervised semantic parsing (2013).

[7] R. Catherine and W. Cohen, Personalized recommendations using knowledge graphs: A probabilistic logic programming approach, in: *Proceedings of the 10th ACM Conference on Recommender Systems*, ACM, 2016, pp. 325–332.

[8] L. Dietz, A. Kotov and E. Meij, Utilizing knowledge graphs for text-centric information retrieval, in: *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, ACM, 2018, pp. 1387–1390.

[9] H. Sun, B. Dhingra, M. Zaheer, K. Mazaitis, R. Salakhutdinov and W.W. Cohen, Open domain question answering using early fusion of knowledge bases and text, *arXiv preprint arXiv:1809.00782* (2018).

[10] X. Wang, P. Kapanipathi, R. Musa, M. Yu, K. Talamadupula, I. Abdelaziz, M. Chang, A. Fokoue, B. Makni, N. Mattei et al., Improving natural language inference using external knowledge in the science questions domain, in: *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33, 2019, pp. 7208–7215.

[11] K. Marino, R. Salakhutdinov and A. Gupta, The more you know: Using knowledge graphs for image classification, *arXiv preprint arXiv:1612.04844* (2016).

[12] M. Allamanis, E.T. Barr, P. Devanbu and C. Sutton, A Survey of Machine Learning for Big Code and Naturalness, *ACM Comput. Surv.* **51**(4) (2018), 81:1–81:37. doi:10.1145/3212695.

[13] U. Alon, O. Levy and E. Yahav, code2seq: Generating Sequences from Structured Representations of Code, in: *International Conference on Learning Representations*, 2019. https://openreview.net/forum?id=H1gKYo09tX.

[14] M. Dumontier, C.J. Baker, J. Baran, A. Callahan, L. Chepelev, J. Cruz-Toledo, N.R. Del Rio, G. Duck, L.I. Furlong, N. Keath et al., The Semanticscience Integrated Ontology (SIO) for biomedical research and knowledge discovery, *Journal of biomedical semantics* **5**(1) (2014), 14.

[15] I. Abdelaziz, K. Srinivas, J. Dolby and J. McCusker, A Demonstration of CodeBreaker: A Machine Interpretable Knowledge Graph for Code, in: *Proceedings of the 19th International Semantic Web Conference (ISWC2020) (Demonstration Track)*, 2020.

[16] D. Engler, D.Y. Chen, S. Hallem, A. Chou and B. Chelf, Bugs as deviant behavior: a general approach to inferring errors in systems code, *SIGOPS Oper. Syst. Rev.* **35**(5) (2001), 57–72. doi:10.1145/502059.502041.

[17] B. Livshits and T. Zimmermann, DynaMine: finding common error patterns by mining software revision histories, *SIGSOFT Softw. Eng. Notes* **30**(5) (2005), 296–305. doi:10.1145/1095430.1081754.

[18] D. Reimer, E. Schonberg, K. Srinivas, H. Srinivasan, B. Alpern, D.R. Johnson, A. Kershenbaum and L. Koved, SABER: smart analysis based error reduction, *ISSTA '96 Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis* (2004), 243–251.

[19] M. Martin, B. Livshits and M. Lam, Finding Application Errors and Security Flaws Using PQL: A Program Query Language, 2005, pp. 365–383. doi:10.1145/1094811.1094840.

[20] C. Hsiao, M.J. Cafarella and S. Narayanasamy, Reducing MapReduce Abstraction Costs for Text-centric Applications, in: *43rd International Conference on Parallel Processing, ICPP 2014, Minneapolis, MN, USA, September 9-12, 2014*, 2014, pp. 40–49. doi:10.1109/ICPP.2014.13.

[21] M. Allamanis, M. Brockschmidt and M. Khademi, Learning to Represent Programs with Graphs, in: *ICLR*, OpenReview.net, 2018.

[22] P. Fernandes, M. Allamanis and M. Brockschmidt, Structured Neural Summarization, *CoRR* **abs/1811.01824** (2018).

[23] M. Bruch, M. Monperrus and M. Mezini, Learning from Examples to Improve Code Completion Systems, in: *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, ACM, New York, NY, USA, 2009, pp. 213–222. ISBN 978-1-60558-001-2. doi:10.1145/1595696.1595728.

[24] S. Proksch, J. Lerch and M. Mezini, Intelligent Code Completion with Bayesian Networks, *ACM Trans. Softw. Eng. Methodol.* **25**(1) (2015), 3:1–3:31. doi:10.1145/2744200.

[25] A.T. Nguyen and T.N. Nguyen, Graph-based Statistical Language Model for Code, in: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 858–868. ISBN 978-1-4799-1934-5. http://dl.acm.org/citation.cfm?id=2818754.2818858.

[26] T.T. Nguyen, H.A. Nguyen, N.H. Pham, J.M. Al-Kofahi and T.N. Nguyen, Graph-based Mining of Multiple Object Usage Patterns, in: *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, ACM, New York, NY, USA, 2009, pp. 383–392. ISBN 978-1-60558-001-2. doi:10.1145/1595696.1595767.

[27] W.U. Ahmad, S. Chakraborty, B. Ray and K.-W. Chang, A Transformer-based Approach for Source Code Summarization, *arXiv preprint arXiv:2005.00653* (2020).

[28] S. Iyer, I. Konstas, A. Cheung and L. Zettlemoyer, Summarizing source code using a neural attention model, in: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 2073–2083.

[29] U. Alon, M. Zilberstein, O. Levy and E. Yahav, Code2Vec: Learning Distributed Representations of Code, *Proc. ACM Program. Lang.* **3**(POPL) (2019), 40:1–40:29. doi:10.1145/3290353.

[30] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis and M. Brockschmidt, Codesearchnet challenge: Evaluating the state of semantic code search, *arXiv preprint arXiv:1909.09436* (2019).

[31] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang et al., Codebert: A pre-trained model for programming and natural languages, *arXiv preprint arXiv:2002.08155* (2020).

[32] D.O. Kubitza, M. Böckmann and D. Graux, SemanGit: A linked dataset from git, in: *International Semantic Web Conference*, Springer, 2019, pp. 215–228.

[33] M. Atzeni and M. Atzori, CodeOntology: RDF-ization of source code, in: *International Semantic Web Conference*, Springer, 2017, pp. 20–28.

[34] A. Jiomekong, G. Camara and M. Tchuente, Extracting ontological knowledge from Java source code using Hidden Markov Models, *Open Computer Science* **9**(1) (2019), 181–199.

[35] K. Cao and J. Fairbanks, Unsupervised Construction of Knowledge Graphs From Text and Code, *arXiv preprint arXiv:1908.09354* (2019).