A Demonstration of CodeBreaker: A Machine Interpretable Knowledge Graph for Code

Ibrahim Abdelaziz, Kavitha Srinivas, Julian Dolby
IBM Research, IBM T.J. Watson Research Center
{ibrahim.abdelaziz1, kavitha.srinivas}@ibm.com, dolby@us.ibm.com

James P. McCusker
Rensselaer Polytechnic Institute (RPI)
{mccusj2}@rpi.edu

ABSTRACT

Knowledge graphs have been extremely useful in powering diverse applications in search, natural language understanding, and image classification. CodeBreaker attempts to construct machine interpretable knowledge graphs about program code to similarly power diverse applications such as code search, code understanding, refactoring, bug detection, and code automation. We have built such a 1.98 billion edges knowledge graph by a detailed analysis of function usage in 1.3 million Python programs in GitHub, documentation about the functions in 2300+ modules, forum discussions on StackExchange and StackOverflow with more than 47 million posts, class hierarchy information, etc. In this work, we will demonstrate one application of this knowledge graph, which is a code recommendation engine for programmers within an IDE. All user interactions within the application get translated into SPARQL queries, which have quite different characteristics than queries against traditional knowledge graphs such as DBpedia or Wikidata. Aspects of code such as data flow are inherently transitive, hence the SPARQL is complex and requires property paths. One of our goals is to provide these queries as a basis for graph querying benchmarks, while allowing users the ability to interact with a real application built on top of a large graph database.

PVLDB Reference Format:
Ibrahim Abdelaziz, Kavitha Srinivas, Julian Dolby. A Demonstration of CodeBreaker: A Machine Interpretable Knowledge Graph for Code. *PVLDB*, 12(xxx): xxxx-yyyy, 2020.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Several knowledge graphs have been constructed in recent years such as DBpedia [6], Wikidata [9], Freebase [4], YAGO [8] and NELL [5]. These graphs now contain vast repositories of knowledge about entities and concepts, and have been successfully used in a number of different application areas such as natural language processing, information retrieval and image classification (see [10] for a comprehensive review of uses of knowledge graphs in applications). With the unprecedented increase of published code libraries in many domains and the growing number of open-source projects, building a knowledge graph for code can be very useful in driving diverse applications around programming such as code search, code automation, refactoring, bug detection, code optimization, etc.

In CodeBreaker, we use a set of generic techniques to construct the first large-scale knowledge graph for code. Specifically, we developed knowledge graph with 1.98 billion edges from deep analysis of 1.3 million Python programs on GitHub. This covered 278K functions, 257K classes and 5.8M methods from 2300+ Python modules, 45M StackOverflow posts and 2.7M StackExchange posts.

One main challenge for building such a knowledge graph is about representation. Existing applications targeting code have broadly focused on one of two approaches (a) treating code as if it were natural language, and applying natural language models and techniques to represent code, (b) parsing code into abstract syntax trees (ASTs), which reflect largely the syntactic structure of the code, and embellishing it when needed with limited types of deeper semantic information such as data flow or program dependence to support the more complex tasks (see [3] for a comprehensive review). Both of these approaches focus on fairly local connections: textual approaches are limited to a handful of words, and ASTs to a few expressions or statements. In CodeBreaker, on the other hand, we represent the code in a more global way than in prior work, i.e., in terms of data flow and control flow. This allows us to analyze connections throughout programs. Specifically, we capture the following: (a) which objects get passed as arguments to which methods, (b) which objects get used to invoke methods (data flow) and (c) which methods get called before which other ones (control flow).

However, global connections make scalable querying over large graphs harder. Querying data- and control-flow is inherently transitive, often naturally expressed as transitive closure. We facilitate writing such queries using SPARQL 1.1 [2], a declarative query language that naturally supports graph patterns, filtering, transitive closure, and has scalable implementations for large graphs. We illustrate how this works in our demo examples.

Also fundamental to this type of knowledge graph is that its really a graph composed of 1.3 million individual graphs, one for each program. Therefore, we use the Resource Description Framework [1] with its support for graphs as our

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

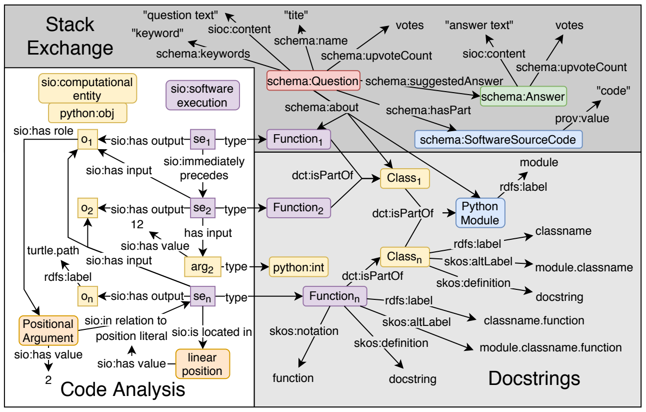


Figure 1: Schema for the code knowledge graph

storage representation. The graphs are however, connected by the specific methods they call, qualified by the library name. In addition, because many important higher level semantic details about the code reside in natural language for human consumption, we linked our knowledge graph to natural language from usage documentation, and forums, using IR techniques. Our knowledge graph is therefore composed of two key conceptual elements: (1) several graphs of data flow and control flow edges between functions, based on analysis of each program we could find on GitHub, (2) a graph of each function or class being connected to its documentation, forum discussions, or its class hierarchy.

In this demo, we demonstrate the use of such a linked knowledge graph, by describing CodeBreaker, a recommendation engine we built into an IDE. Our key focus here is on the sorts of storage and query support needed for use cases of code knowledge graphs. The conference audience will be able to interact with CodeBreaker through a graphical interface, where they can find 1) the next most likely call given the current method call, 2) popular data science pipelines used by others that are similar to their own, 3) relevant forum discussions based on their own code[1]. For each of these scenarios, we provide SPARQL queries, which can provide the basis for graph querying benchmarks, along with the knowledge graph.

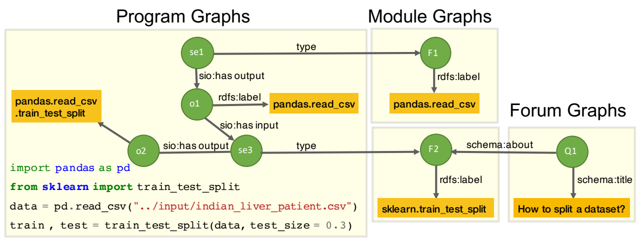
2. OVERVIEW OF CODEBREAKER

In this paper, we focus on the schema, data representation and graph queries that drives CodeBreaker. For details on the construction of the knowledge graph and the linking, see [7].

2.1 Schema

Figure 1 shows the schema for the knowledge graph. Nodes in the graph of the type `Software Execution` refer to nodes generated by the static analysis code, one per each method call in program code. These nodes are linked by the `type` property to corresponding methods, classes and functions, when possible, using traditional information retrieval techniques. Forum posts are broken up into the question, the answers, and each of these in turn was parsed to separate

[1]Video of the demo and the underlying knowledge graph are available at `http://graph4code.whyis.io/download/`



```
import pandas as pd
from sklearn import train_test_split
data = pd.read_csv("../input/indian_liver_patient.csv")
train , test = train_test_split(data, test_size = 0.3)
```

Figure 2: Data representation in the code knowledge graph

out the textual components and code components. Forum posts are linked by `schema.org:about` property to the relevant method, class and function nodes.

2.2 Data Representation

Figure 2 shows the modeling choices in knowledge graph. Each program results in a specific graph within the RDF format, as shown by the yellow rectangles. Each module has its own RDF graph, with documentation about the module's classes, functions and methods. Each StackExchange or StackOverflow question and its associated answers is in a separate RDF graph. We note that the heavy use of individual graphs for modeling documentation strings and forum posts was guided by two reasons (a) to accomplish querying efficiency, because most RDF stores index data by graphs, and (b) to better manage the graph data, making it easy to add and remove graphs in the future.

3. DEMONSTRATION OVERVIEW

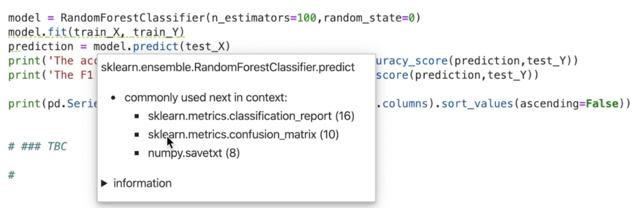
The objective of this demo is to show SIGMOD attendees how CodeBreaker is created and showcase its usefulness and promise. In this demonstration, we will provide a complete interface to show the potential of CodeBreaker in coding assistance. In particular, we integrated CodeBreaker with Jupyter Lab[2].

In this section, we describe the different demonstration scenarios with which users can interact with CodeBreaker. In particular, we demonstrate the following functionalities: 1) at any step of the code, show the most commonly used next steps, 2) Across all indexed code, show similar flows to the current code; users can take advantage of this to see how others tend to construct data science pipelines. 3) find relevant user forums in StackOverflow and StackExchange; with the search taking into context the semantics of the user's code. We also show how these functionalities are implemented and the SPARQL queries that were used to get the corresponding information from the knowledge graph.

3.1 Next Coding Step

The first scenario involves a developer in an IDE being provided 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

[2]This was made straightforward using Jupyter Lab's Language Server Protocol support, `https://github.com/krassowski/jupyterlab-lsp`



```
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

Figure 3: Finding most commonly used next step

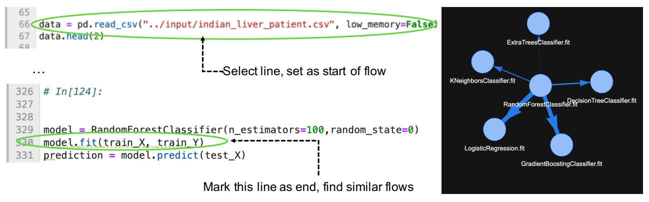
that constructed the classifier. Figure 3 shows an example of a real Kaggle notebook, where users can select any expression in the code and get a list of the most common next steps along with the frequency with which the next step is observed. For example, in similar contexts after a `model.predict` 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 call, but the call is very helpful to understand the properties of a classifier. We show in Figure 4 the SPARQL query used to support this use case. The query locates the *predict* call of *RandomForestClassifier*, collects all calls that follow it and ranks them by frequency. The filter constructs ensure that only functions calls are retrieved and not their superclasses. Note that each data flow step consists of a two-edge path, `sio:SIO_000230/^sio:SIO_000229`.

```
select (count(?g) as ?c) ?predict ?prev_type where {
 graph ?g {
   ?prev a  py:sklearn.ensemble.RandomForestClassifier .
#  ?prev ('has input'/^'has output')+ ?equiv.
   ?prev sio:SIO_000229/^sio:SIO_000230 ?equiv .
   ?equiv a
    py:sklearn.ensemble.RandomForestClassifier.predict .
   ?equiv sio:SIO_000229/^sio:SIO_000230 ?next .
   ?next a ?predict .
   filter( ?predict != prov:Activity
#        && ?classifierName != 'software execution'
         && ?predict != sio:SIO_000667
         && ?predict != py:print
         && ?predict != graph4code:Function)
 }
} group by ?predict ?prev_type order by desc(?c) limit 3
```

Figure 4: Query to find the next step in a program

3.2 Get Similar Flows From Other Programs

The second scenario helps data scientists understand data science pipelines similar to the ones they have written, and use this to understand what types of models other data scientists use given similar code context. As shown in Figure 5, to define the pipeline, the developer needs to choose two steps in the pipeline; such as from the point



```
data = pd.read_csv("../input/indian_liver_patient.csv", low_memory=False)
data.head(2)
```

...

```
# In[124]:


model = RandomForestClassifier(n_estimators=100,random_state=0)
model.fit(train_X, train_Y)
prediction = model.predict(test_X)
```

Select line, set as start of flow
Mark this line as end, find similar flows
ExtraTreesClassifier.fit
KNeighborsClassifier.fit
RandomForestClassifier.fit
DecisionTreeClassifier.fit
LogisticRegression.fit
GradientBoostingClassifier.fit

Figure 5: Finding similar data science pipelines

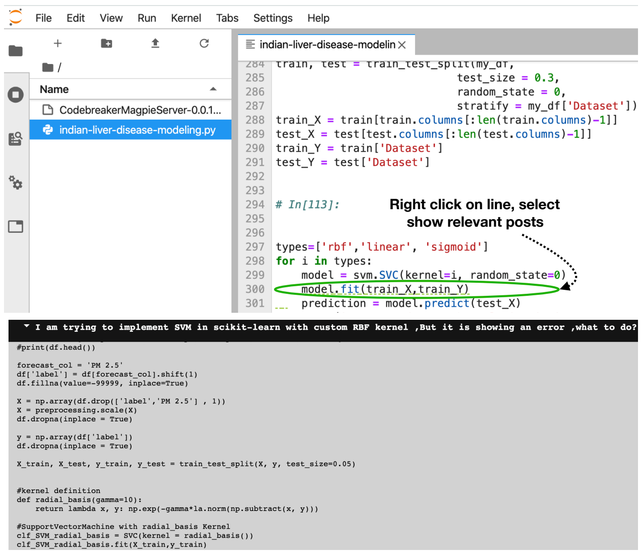
a dataset is read (e.g. `read_csv`) until a fit call is performed (e.g., `model.fit`). In the example program, data flows from `read_csv` to `RandomForestClassifier.fit`. The interface allows users to query what other classifiers tend to be invoked on the same data as `RandomForestClassifier`. On the right side of Figure 5, we can see that in Kaggle notebooks, people tend to use `RandomForestClassifier`, along with `Gradient Boost Classifier` and `K neighbours Classifier` to fit the same data. The thickness of the arrows denote how frequently these classifiers have been used together. This recommendation gives data scientists options of different classification models to try.

```
select (count(?g) as ?c) ?classifierName where {
 graph ?g {
  ?read a py:pandas.read_csv .
# ?fit1 ('has input'/^'has output')+ ?read.
  ?fit1 (sio:SIO_000230/^sio:SIO_000229)+ ?read.
  ?fit1 rdfs:label ?fit1Label.
  filter(regex(?fit1Label, "fit$"))
  ?fit1 sio:SIO_000230/^sio:SIO_000229 ?rfc.
  ?rfc a py:sklearn.ensemble.RandomForestClassifier.
  ?rfc sio:SIO_000228 [ # has role
    sio:SIO_000300 "0"^^<xsd:integer>; # has value
    sio:SIO_000668 ?fit1 # in relation to
  ].

  ?fit2 (sio:SIO_000230/^sio:SIO_000229)+ ?read.
  ?fit2 rdfs:label ?fit2Label.
  filter(regex(?fit2Label, "fit$"))
  ?classifier sio:SIO_000228 [ # has role
      sio:SIO_000300 "0"^^<xsd:integer>; # has value
      sio:SIO_000668 ?fit2 # in relation to
  ].
  ?classifier a ?classifierName.
  minus {
    ?classifier a py:sklearn.ensemble.RandomForestClassifier
  }
  filter( ?classifierName != prov:Activity
#       && ?classifierName != 'software execution'
        && ?classifierName != sio:SIO_000667
        && ?classifierName != graph4code:Function)
 }
} group by ?classifierName order by desc(?c)
```

Figure 6: Compute data path edges from data science inputs to uses across all programs, ordering by the most common

Figure 6 shows the query to gather data science pipelines similar to the one the user has written. Given an input expression *read*, in this case `pandas.read_csv`, we track how that data is used, in this case in `RandomForestClassifier.fit` calls, *fit*1. To find similar flows, we track flows from the same read to other `fit` calls, *fit*2, using `minus` to filter the original classifier.


File
Edit
View
Run
Kernel
Tabs
Settings
Help
indian-liver-disease-modelin
Name
CodebreakerMagpieServer-0.0.1...
indian-liver-disease-modeling.py
Right click on line, select show relevant posts

```
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)
    prediction = model.predict(test_X)
```

I am trying to implement SVM in scikit-learn with custom RBF kernel ,But it is showing an error ,what to do?

```
#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)
```

Figure 7: Finding relevant forum posts

Note that this query taxes many aspects SPARQL, as we need multiple transitive property paths to capture $fit1$ and $fit2$, multiple regex filters, aggregation and a `minus`.

3.3 Get Relevant Forums Posts

In a third scenario, a developer can retrieve forum posts in StackOverflow and StackExchange relevant to the code written so far. Figure 7 shows how one can show relevant forum posts for the path in the code up to `sklearn.svm.SVC.fit`. The figure shows one of the posts. Note that the code written in the Kaggle notebook has data flowing from a `read_csv` to a `train_test_split` to a `SVC.fit`. This exact flow exists in the retrieved StackOverflow post which also a `read_csv` to a `train_test_split` to a `fit` on an `SVC`. Similarities in flow are useful because it implies that the post is discussing the same coding context. It is hard to detect code similarity at a token level since the SVC object is called `model` in the Kaggle notebook and `clf_SVM_radial_basis` in this forum post. Since CodeBreaker's graph decomposes the Kaggle code into its semantics, we can take each of the nodes in the dataflow for the Kaggle program and issue a SPARQL query which then gathers up relevant StackOverflow posts. This functionality is achieved by the query in Figure 8 which tries to get forum posts about (`schema:about`) `SVC` and `SVC.fit`. The retrieved results include title, content and the suggested answer of each question, if any. The query also orders the forum posts according to the frequency of links made from each call in to its StackOverflow posts.

4. CONCLUSION

In this paper, we demonstrated the usability of the CodeBreaker knowledge graph in code assistance within an IDE. In particular, we show how one can query the knowledge graph using SPARQL for code suggestion, finding similar data science pipelines and context-based search in web forums. The knowledge graph is extensible and we made it publicly available to the larger community for use.

```
select ?t ?q ?aa ?c ?v ?a where {
  {
    select ?v (count(?tp) as ?c) where {
      ?v schema:about ?tp.
    } group by ?v
    values (?tp) {
      (py:sklearn.svm.SVC)
      (py:sklearn.svm.SVC.fit)
    }
  }
  ?v schema:name ?t ;
     sioc:content ?q .
  optional {
    ?v schema:suggestedAnswer ?a .
    ?a sioc:content ?aa .
  }
} order by desc(?c)
```

Figure 8: Query to retrieve StackOverflow and StackExchange posts that use or mention the scikit-learn SVC learning method.

5. REFERENCES

[1] Resource Description Framework (RDF). https://www.w3.org/TR/rdf-primer/.

[2] SPARQL 1.1. https://www.w3.org/TR/sparql11-query/.

[3] 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):81:1–81:37, July 2018.

[4] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *In SIGMOD Conference*, pages 1247–1250, 2008.

[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, pages 1306–1313. AAAI Press, 2010.

[6] 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):167–195, 2015.

[7] K. Srinivas, I. Abdelaziz, J. Dolby, and J. P. McCusker. Graph4code: A machine interpretable knowledge graph for code. *arXiv preprint arXiv:2002.09440*, 2020.

[8] 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, pages 697–706, New York, NY, USA, 2007. ACM.

[9] D. Vrandečić and M. Krötzsch. Wikidata: A free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, Sept. 2014.

[10] Q. Wang, Z. Mao, B. Wang, and L. Guo. Knowledge graph embedding: A survey of approaches and applications. *IEEE Trans. Knowl. Data Eng.*, 29(12):2724–2743, 2017.