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

Ibrahim Abdelaziz[1], Kavitha Srinivas[1], Julian Dolby[1] and James P. McCusker[2]

[1] IBM Research, IBM T.J. Watson Research Center
{ibrahim.abdelaziz1, kavitha.srinivas}@ibm.com, dolby@us.ibm.com
[2] Rensselaer Polytechnic Institute (RPI), mccusj2rpi.edu

Abstract. Knowledge graphs have been extremely useful in powering diverse applications like natural language understanding. CodeBreaker attempts to construct machine interpretable knowledge graphs about program code to similarly power diverse applications such as code search, code understanding, 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 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.

1 Introduction

Several knowledge graphs have been constructed in recent years such as DBpedia [4], Wikidata [6] and Freebase [3]. 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 and information retrieval [7]. 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

Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-2721/paper568.pdf

GitHub. This covered 278K functions, 257K classes and 5.8M methods from 2300+ Python modules, 47M posts from StackOverflow and StackExchange.

One main challenge for building such a knowledge graph is about representation. We represent the code in a more global way than in prior work based on program text or ASTs, 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. This can be achieved using SPARQL 1.1 [2], which naturally supports graph patterns, filtering, transitive closure, and has scalable implementations for large graphs.

CodeBreaker is 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 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 information retrieval techniques. For details on the construction of the knowledge graph, its schema and how different code and documentation are linked, see [5].

We demonstrate CodeBreaker[3] using 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 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. For each of these scenarios, we provide SPARQL queries, which can provide the basis for graph querying benchmarks, along with the knowledge graph.

2 Demonstration Overview

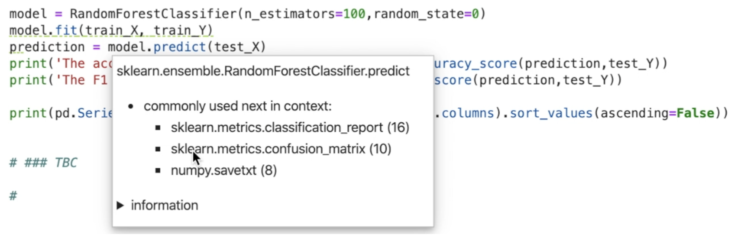
We provide a complete interface to show the potential of CodeBreaker in coding assistance. In particular, we integrated CodeBreaker with Jupyter Lab using its Language Server Protocol support, `https://github.com/krassowski/jupyterlab-lsp`.

2.1 Next Coding Step

We show a developer being provided commonly used next steps, based on the context of their current code. Context means data flow predecessors of the node

[3] A video of the demo is available at `https://github.com/wala/graph4code/blob/master/docs/figures/demo_v2.mp4` while the underlying knowledge graph is available at `https://wala.github.io/graph4code/`

of interest; we take a simple example of the single predecessor call that constructed the classifier. Figure 1 shows a real Kaggle notebook, where users can select any expression in the code and get the most common next steps along with their frequencies. In similar contexts, 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 this example, 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. The exact SPARQL query to support this functionality is available `https://github.com/wala/graph4code/blob/master/usage_queries/find_next_step.sparql`.



```
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

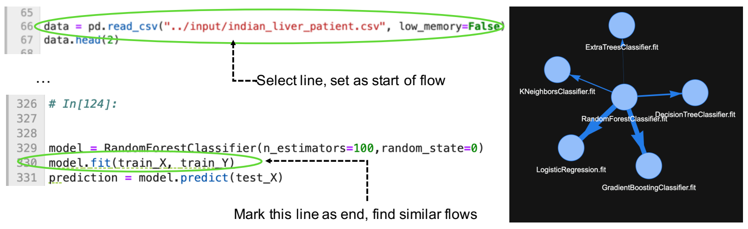
- commonly used next in context:
  - sklearn.metrics.classification_report (16)
  - sklearn.metrics.confusion_matrix (10)
  - numpy.savetxt (8)

▶ information

Fig. 1. Finding most commonly used next step

2.2 Get Similar Flows From Other Programs

The second scenario helps understand data science pipelines similar to existing ones, and use this to understand what types of models other data scientists use given similar code context. As shown in Figure 2, to define the pipeline, the developer needs to choose two steps in the pipeline; such as from the point 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 2, 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. The SPARQL query for gathering the relevant data science pipelines can be found in `https://github.com/wala/graph4code/blob/master/usage_queries/find_similar_flows.sparql`. This query taxes many aspects SPARQL, as we need multiple transitive property paths to capture the fit calls, multiple regex filters, aggregation and a `minus`.



```
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

Fig. 2. Finding similar data science pipelines

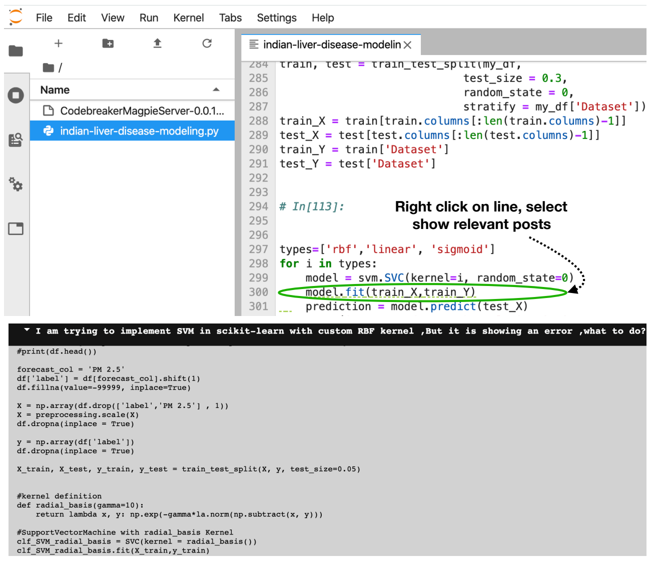
2.3 Get Relevant Forums Posts

A developer can find posts in StackOverflow and StackExchange based on code written so far. From the code, CodeBreaker finds the forum posts with similar flows. Similarities in flow are useful because it implies that the post is discussing the same coding context. However, it is hard sometimes to detect code similarity at a token level since the same object can be called differently. 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.

Figure 3 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. The corresponding SPARQL query can be found at `https://github.com/wala/graph4code/blob/master/usage_queries/find_stack_overflow_posts.sparql`.

3 Conclusion

In this paper, we demonstrated the use of CodeBreaker, the first large-scale knowledge graph for code, in code assistance within an IDE. In particular, we show how one can get code suggestions, finding similar data science pipelines


File
Edit
View
Run
Kernel
Tabs
Settings
Help
/
Name
CodebreakerMagpieServer-0.0.1...
indian-liver-disease-modeling.py
indian-liver-disease-modelin
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)
```

Fig. 3. Finding relevant forum posts

and context-based search in web forums. The knowledge graph is extensible and we made it publicly available to the larger community for use.

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. Bollacker, K., Evans, C., Paritosh, P., Sturge, T., Taylor, J.: Freebase: a collaboratively created graph database for structuring human knowledge. In: In SIGMOD Conference. pp. 1247–1250 (2008)
4. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., Bizer, C.: DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. Semantic Web Journal **6**(2), 167–195 (2015)
5. Srinivas, K., Abdelaziz, I., Dolby, J., McCusker, J.P.: Graph4code: A machine interpretable knowledge graph for code. arXiv preprint arXiv:2002.09440 (2020)
6. Vrandečić, D., Krötzsch, M.: Wikidata: A free collaborative knowledgebase. Commun. ACM **57**(10), 78–85 (Sep 2014). https://doi.org/10.1145/2629489
7. Wang, Q., Mao, Z., Wang, B., Guo, L.: Knowledge graph embedding: A survey of approaches and applications. IEEE Trans. Knowl. Data Eng. **29**(12), 2724–2743 (2017)