CECS

CENTER FOR EMBEDDED & CYBER-PHYSICAL SYSTEMS

Deep Code Curator – code2graph Part-II

Shih-Yuan Yu

Arnav Vaibhav Malawade

Aung Myat Thu

Mohammad Abdullah Al Faruque

Center for Embedded and Cyber-Physical Systems

University of California, Irvine

Irvine, CA 92697-2620, USA

{shihyuay, malawada, amthu, alfaruqu}@uci.edu

CECS Technical Report TR# 20-01

April 30, 2020

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00111990010

Contents

1 Introduction

Deep Learning (DL) is a fast-growing field with thousands of researchers working on new models, architectures, and algorithms every day. Usually, DL models require experts to spend their time reading through the state-of-the-art publications, making sense of the claims and the results, and using their experience to create links between the body of knowledge to validate new ideas. However, as deep learning algorithms and implementations across various fields grow at an explosive rate, keeping up with all the latest publications and source code becomes a challenge for researchers and practitioners who hope to contribute their work to the community of deep learning.

As a result, the goal of the project - **Deep Code Curator (DCC)** - aims to tackle this issue by utilizing the information from scientific publications and accompanied software implementations of architectures and methodologies. The DCC collects the scientific papers and represents them in a universal representation that can dramatically decrease time, effort, and resources spent while curating existing DL literature and algorithms. In DCC, we focus on three primary modalities of a scientific paper that can provide useful information: text, images, and source code, with each supported by modules text2graph, image2graph, and code2graph, respectively. The main goal of these modules is to create the Resource Description Framework (RDF) knowledge graphs that contain necessary information about publications. These RDFs are then aligned and merged into a supergraph, which serves as the architectural representation of a scientific paper and can be used to explore and compare the publications across various fields that share similar DL architectures.

This technical report focuses on reporting the status of one DCC's module: code2graph. The primary goal of code2graph is to extract RDF graphs from code implementations associated with DL scientific papers. We first focus on the source code using a well-known DL framework, especially TensorFlow or Keras. These generated graphs serve to complete the architectural information in the supergraph of DCC partially. Besides, these RDF graphs enable the training of stochastic code inference models to reverse-engineer the source code by utilizing the traditional rule-based inference methodologies and the information from knowledge graphs. In code2graph, extracting knowledge graphs from code poses several challenges:

1. The knowledge graphs generated from the code need to encompass necessary information to be converted back to code during graph2code while still being abstract enough to be aligned with the knowledge graphs generated by other modules in DCC.
2. The knowledge graphs must be abstracted across all the scientific source code, which might use different software libraries, development environments, and coding styles for the downstream inference tasks.

During **DCC Phase 1**, we explored the methodologies to convert source code into RDF graphs. First, we primarily focused on the implementations that use the TensorFlow runtime library because it is a cross-platform framework where all the computations get converted into a data-flow graph automatically. With this, we developed a pipeline called **Computational Graph-Based Approach** to convert source code into such a graphical form, which allows us to preserve the flow of values in DL models. However, **in Milestone 5**, we also discovered that the full automation of preprocessing in this approach is quite challenging as the developers can have different programming styles or make assumptions in their repositories. This finding,

in turn, makes it hard to successfully identify the way to initiate the programs and acquire the desired TensorFlow graphs in the runtime. Therefore, we also explored an alternative approach, called the **Lightweight Approach**, that can statically extract the knowledge graphs by analyzing the syntactic structure of source code. This method can generate RDF graphs without triggering the programs, thus being more scalable in creating a larger dataset from source code. During **DCC Phase 2**, **in Milestone 7**, we have also helped the teams add code-related definitions to a universal ontology shared across the DCC project, and used the ontology to update lightweight graphs generated from code2graph. Beginning from **Milestone 8**, we have formulated **Graph Encoding** proceses for evaluating the RDF graphs and applied graph embedding methods. By **Milestone 9**, we have explored three different approaches.

In this report, we will focus on the progress in DCC Phase 2, and only summarize the parts covered in our previous report [1]. The rest of this report is structured as follows: In Section 2, we will describe the two graph extraction approaches and three graph embedding methods used for evaluating RDF graphs; we will also introduce the implementations of pipeline for each approach. In Section 4, we will demonstrate the results that we delivered throughout the DCC project. Finally, in Section 5, we will conclude our insights and learned lessons from working with the code2graph module in the DCC project and potential future directions. All of the associated codebases[1] are open-source in the GitHub community.

2 Proposed Methodology

Figure 1 illustrates the pipeline for Knowledge Graph Extraction. Initially, in Milestone 1 and 2, we focused on extracting the graphs from code through the **Computational Graph-Based Approach** (shown in green-colored boxes). Starting from Milestone 3, we also explored how to convert code into graphs using the information on its Abstract Syntax Trees (ASTs) which is a common intermediate representation used in compilers for code optimization. Analyzing the ASTs of the programs in each DL publication, we construct the graphs that track API call occurrence and their dependencies. We name this alternative the **Lightweight Approach** (shown in blue-colored boxes).

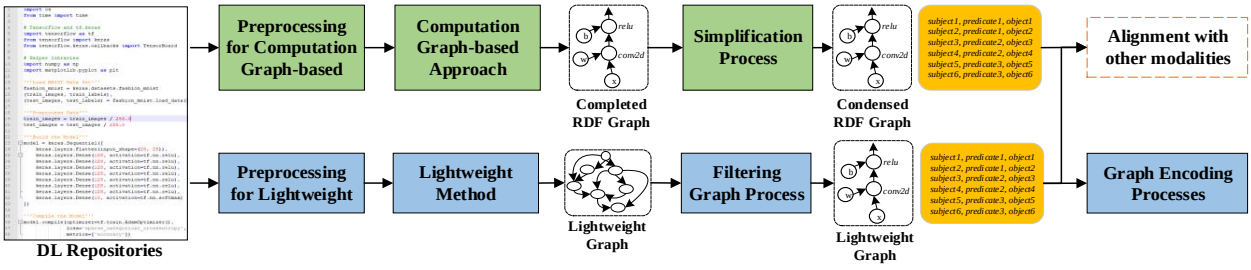

Figure 1: The Knowledge Graph Extraction pipeline architecture approaches in code2graph.

[1] https://github.com/deepcurator/DCC/tree/master/src/code2graph

Apart from graph extraction approaches, we have also explored graph learning algorithms that evaluate the knowledge graphs produced from code2graph to spot similar papers better, determine originality, and ultimately perform code reconstruction. Initially, we target on representing the source code or the code elements (ex. functions) using low-dimensional latent vectors (embeddings). To learn the representations, we investigated various approaches that treat the code elements or code as text documents, abstract syntactic trees, or, the most generic form, knowledge graphs.

2.1 The Computation Graph-Based Approach

The goal of this approach is to extract the computation graphs from the source code implementations associated with DL publications. As mentioned in Section 1, we focused on the codebase that uses TensorFlow[2] as its programming framework because TensorFlow adopts a data-flow programming model for better runtime performance, allowing us to extract the graphical form of computation along its pipeline. Practically, a client program written developers will initially get converted into a computation graph. TensorFlow operates with computation graphs using a serialized graph definition, called Protocol Buffer. Finishing this conversion, TensorFlow then creates a Session for executing only the necessary computations (lazy programming) in a distributed manner on the CPUs and GPUs available in the host system. A TensorFlow data-flow graph (computation graph) consists of nodes, that represent operations, and edges that represent the control dependencies among various nodes. Figure 2 shows a program example of performing a matrix multiplication over two constants and its data-flow graph visualized by TensorBoard. This data-flow graph consists of two constant operators are responsible for creating constant values, and a MatMul operator takes the outputs from Constant operators and calculates the multiplying result.

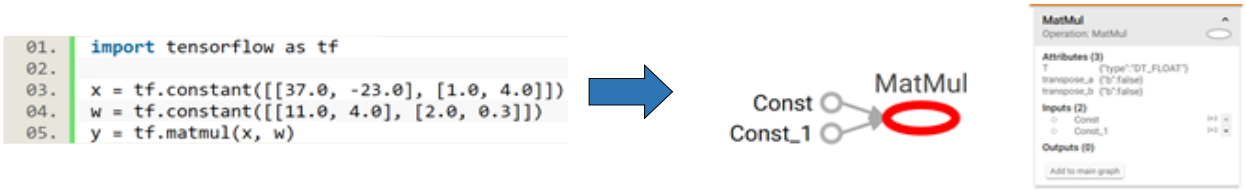

Figure 2: A matrix multiplication program example in Tensorflow and its converted data-flow graph in Tensorboard.

2.1.1 Data Preprocessing

To automate the computation graph extraction, we designed a pipeline to preprocess code repositories, which includes: **Collecting all non-TensorFlow repositories**, **Creating the virtual environment**, **Resolving the package requirements**, **Identifying the main program**, and ultimately **Injecting the code snippet**. We will explain each step subsequently.

[2] https://www.tensorflow.org/

Collecting all TensorFlow-based Repositories: the sources such as zziz[3] and paperswithcode[4] contain the published papers with corresponding available open-source repositories. First, we need to check if they use programming frameworks such as TensorFlow and Keras[5]. However, at the beginning of DCC, Siemens provided a dataset that includes 100 papers (100-dataset) with source code where only 16 of them use TensorFlow. Compared to commonly-seen knowledge bases (FreeBase15K, FreeBase15K-237, etc…), the number of triples that we acquired from this dataset is far from enough to perform meaningful downstream tasks. Therefore, to get a larger dataset, we have created a web crawler (PWCscraper) to scrape paperswithcode.com automatically. Also, we made PWCscraper a continuous service so that the dataset can grow larger daily. The metadata that PWCscraper includes title, abstract, the link of the article, the link of code, tags, and along with the compressed source code from its master branch. Figure 3 is a screenshot of a data item on paperswithcode.com. The tags, such as Data Augmentation or Semi-supervised image classification or text classification in Figure 3, are labeled by paperswithcode community members and can potentially assist in higher semantic inference tasks.

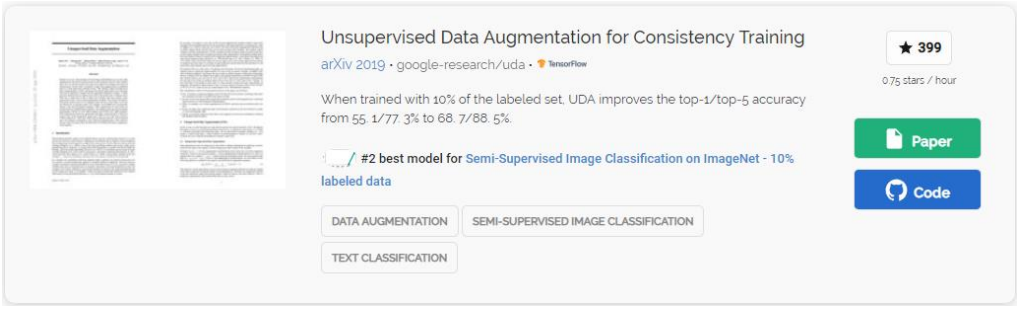

Figure 3. The screenshot of a data item on paperwithcode.com

Creating the Virtual Environment: the code repositories in the dataset might be using Python 2 or 3. To create a virtual environment, we first have to identify the Python version used in the code. To do this, we incorporated a Python package called py_compile[6], which is useful in sharing the Python modules without revealing the source code. In our context, we used py_compile to pre-check the source files and verify the syntax. On a system where Python 2 and 3 coexist, we utilized this pre-check function to determine the Python version used in each repository. Then, we can use this information to create a virtual environment accordingly.

Resolving the requirements: after acquiring the virtual environment, we also need to automate the process of installing the necessary packages for each code repository. Often, a list of requirements is listed in requirements.txt file. In this case, running the command "pip install requirements.txt" can automatically

[3] https://github.com/zziz/pwc

[4] https://paperswithcode.com/

[5] https://keras.io/

[6] https://docs.python.org/3/library/py_compile.html

install the required packages. If such a requirement file does not exist in the repository, we developed an AST walker that analyzes all the "import" and "import from" instructions by parsing the AST of the code. These nodes can be parsed to find out the necessary packages and searched in the Python package index (https://pypi.org/pypi/[package name]/json) to get the detailed installation information.

Code Injection: to extract the computation graph from the source code, the code needs to be partially compiled up-to-the point where the necessary data-flow graphs have been appended to the Tensorflow graph instance. In Keras, the developers usually call a "compile" method before the training starts. For native TensorFlow APIs, developers trigger sess.run() after the computation initialization has finished. In practice, using the ast helper function NodeVisitor, we can search and examine each Call instance in the code and verify whether its function name is "compile" or "run". Once found, we then modify the ASTs by programmatically injecting a small snippet (that uses the Writer module in TensorFlow) into these locations. Finally, we use the function "to_source" in a Python library astor to regenerate the source code. Once this modified script is triggered, it will automatically extract the serialized graph definitions of the models in code repositories. The small snippet includes acquiring the current TensorFlow session and creating a writer module that directs it to a location we want to store the file. Lastly, the injected snippet forces the program to exit, avoiding the subsequent training process that can dramatically increase the complexity of preprocessing.

2.1.2 Extracting the Computational Graphs

In the previous technical report, we elaborated on computation graph extraction. Here, we will only summarize the general pipeline. With the preprocessed source code, we can execute the program along with the inserted code snippet that utilizes the TensorFlow built-in FileWriter module to export the summary files in Protocol Buffer format. With EventAccumulator, a built-in TensorBoard module, we process these summary files and convert them into GraphDef objects which hold the serialized protocol buffer data structure. Lastly, the GraphDef gets turned into a JSON format that contains enough information about DL architectures. In the pipeline, we developed a JSON-to-RDF Parser to generate the **Complete RDF Graph** using all the information such as code hierarchy, operation types, input-output relations, and names on the JSON object.

Simplification Process: The Complete RDF graphs require a simplification process as they are too complicated to be handled in subsequent processes, as illustrated in Figure 4. In response to this, we have applied a rule-based breadth-first search (BFS) algorithm on Complete graphs. When visiting nodes recursively, the algorithm decides between collapsing the currently visited node or trimming the rooted subgraph based on whether the currently visited node is in the pre-set "unnecessary" or "interesting" node lists. However, the node lists used for simplification were hard-coded based on our general machine learning knowledge, thus being not generic enough to handle every repository in the dataset. One promising future direction here is to generalize the approach so that we can adapt this simplification process to more code implementations.

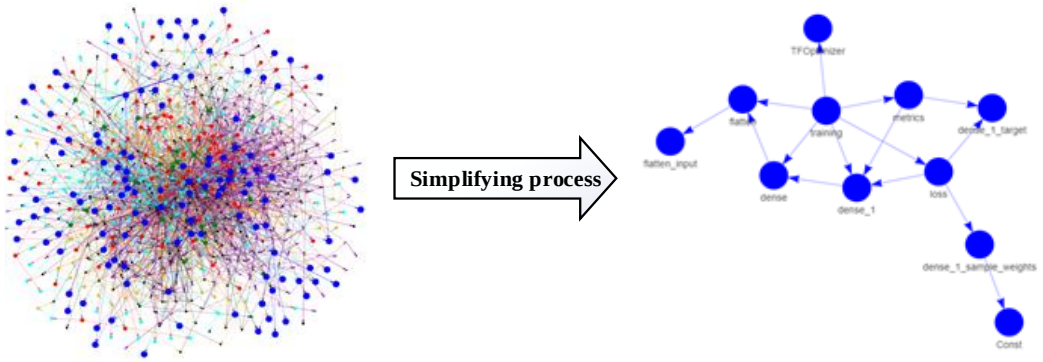

Figure 4. The complete and condense RDF graph for MNIST example.

2.1.3 Challenges on Automating the Preprocessing

In 2.1.1, we have described the each functional components for preprocessing. However, in practice, the repositories are diverse in programming style and have different assumptions, making full automation very challenging. The first challenge is identifying the entry point of the code, which can exist in different files for each repository. In response to this, we tried searching for the files that have the instruction "if __name__ == "__main__":", which gives us a hint that this source file contains a runnable script. Sometimes, "main.py" exists, and includes the main code that executes the model construction, but sometimes multiple scripts can coexist for compiling their models in various flavors. In this case, we still need manual effort on skimming through and understanding the source code and the paper to determine from which model we want to extract the computation graphs. The second challenge is the parameterization issue. Many developers allow the hyperparameters used in their models to be tuned by specifying the arguments when running the scripts. For example, one may like to design the script in this manner "*python -input 6 -output 3 -hidden_layers 5 -activation relu first_model.py*" so that by changing the arguments, they can search for the golden setting for their models. Scanning through the "readme.md" under GitHub repositories might give us some hints on how to run the script. However, some developers might just not include such information. Therefore, it still requires manual effort to find out how to run the script.

With these challenges, it is almost impossible to automate the preprocessing. One possible solution is to record all the tried settings that succeed in invoking the model compilation so that we can reproduce them later. Otherwise, it requires massive manual effort to create a large dataset because of the preprocessing. Therefore, in the following Milestones of the DCC project, we explored alternatives to extract RDF graphs without invoking the computation.

2.2 The Lightweight Approach

Section 2.1.3 describes the challenges of scaling up the dataset in the preprocessing of the Computation Graph-Based Approach as it requires triggering the program to an extent where we can stop and perform the extraction of computation summaries. Therefore, starting from Milestone 3, we also have explored the possibilities of the AST-based graph extraction approach. We call this the Lightweight Approach, which allows us to acquire graphs without actually executing the code. Usually, developers program their models

by invoking TensorFlow or Keras APIs. Therefore, as illustrated in Figure 5, the Lightweight Approach first analyzes the AST of source code to acquire a Function Call Graph. Then, by traversing each function node on this graph, we convert a function call graph into a Lightweight graph. The Lightweight graph captures the occurrence and order of API invocations, which can assist us in reconstructing the computation graph of their models.

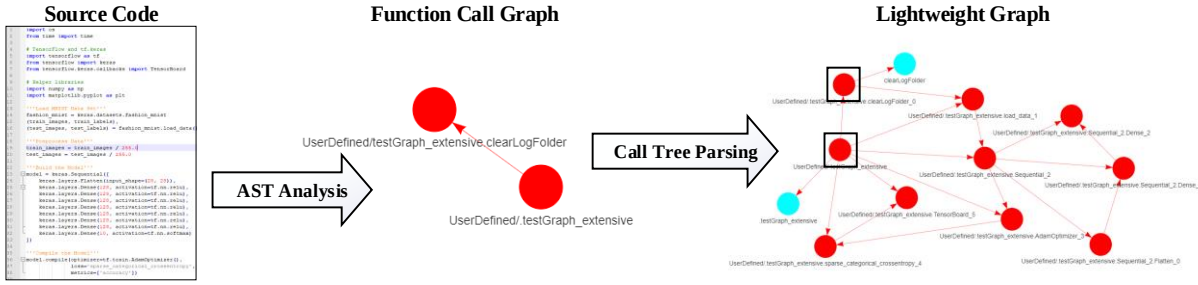

Figure 5. The complete and condense RDF graph for MNIST example.

2.2.1 Data Preprocessing

Even though the Lightweight Approach is a static graph extraction approach, we still need to preprocess the code implementations in the dataset. Since the packages ast[7] and pyan[8] used in the pipeline are only applicable to Python 3, the Lightweight Approach will raise exceptions when running repositories that use Python 2, a mix of Python 2 and Python 3, or have indentation issues by themselves. To overcome these issues, we use two libraries: 2to3[9] and autopep8[10], which can translate the program files from Python 2 to Python 3 and make the indentation consistent throughout the files.

2.2.2 Extracting the Lightweight Graphs

After converting all the program files in the repository to Python 3, we utilized the Python packages ast and pyan to extract the ASTs and to generate the function call graphs. Then, we developed TFCallVisitor to track the TensorFlow API calls by traversing the calling relationships, generating a tree-like structure known as Call Trees that contain both sequential and hierarchical information. We have elaborated on the details of the Lightweight Approach in the previous technical report. As a result, we only describe one of the most critical functional components as below:

Call Tree Parsing: With the function call graphs, we then utilize TFCallVisitor to traverse the function call graphs recursively and to match the name of the function calls with the keywords defined in the ontology. Upon identifying a TensorFlow API invocation, the TFCallVisitor appends information such as

[7] https://docs.python.org/3/library/ast.html

[8] https://github.com/davidfraser/pyan

[9] https://docs.python.org/2/library/2to3.html

[10] https://github.com/hhatto/autopep8

arguments, labels, or keywords to the node. Finally, we generate the RDF triplets according to the attributes and the sequential information that each node carries in the resultant Call Trees.

2.3 The Graph Embedding Approaches

In code2graph, we developed graph learning pipelines to evaluate the RDF graphs generated from both approaches. These methods target representing code elements (ex. functions) using low-dimensional latent vectors, which are also known as embeddings. With the training pipeline, the embeddings of source code capture the semantic properties of an object by representing it with its features distributed across multiple vector components. Figure 6 illustrates the pipeline for embedding tasks. To learn the representations, we investigated various approaches to embed code elements or code by treating them as text documents (doc2vec), abstract syntactic trees (code2vec), or knowledge graphs (pykg2vec). We will then describe these three approaches in the subsequent subsections.

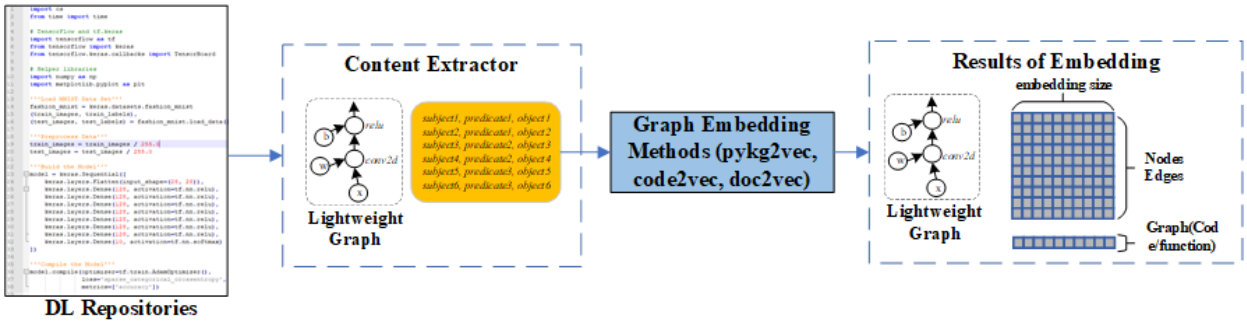

Figure 6. The architecture of performing embedding tasks.

2.3.1 The Baseline Method: Doc2vec

The doc2vec model learns the fixed-length "Paragraph Vectors" to represent the variable-length pieces of texts (such as sentences, paragraphs, and documents). It originates from the word2vec [2] that models the co-occurrence of one specific word and its neighboring words so that the learned embedding for each word can preserve the word's meaning. Figure 7 shows that the doc2vec [3] is an extension of the word2vec model and can learn a latent vector associated with each document on top of the learned word vectors in documents in an unsupervised manner. For code2graph evaluation, we thought this is a good baseline approach that treats the source code or code elements as text pieces. In practice, we used the implementation from the open-source library, genism[11], to learn the latent vectors that represent the functions in source code by treating the content of these functions as pure text pieces. We can easily extend the resolution to learn the embeddings for the source code or even the repository. However, treating the source code or code elements as text pieces will lead to the loss of structural information such as the orders or the relationships between instructions.

[11] https://github.com/RaRe-Technologies/gensim

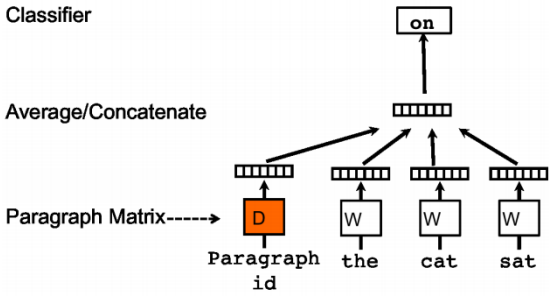

Figure 7. The architecture of the doc2vec model that performs embedding tasks.

To perform preprocessing on dataset for training doc2vec model, we created "Doc2vecDataExtractor" class as follows:

```
1.  class Doc2vecDataExtractor:
2.
3.      ''' this class is for creating dataset for doc2vec '''
4.
5.      def __init__(self, code_path, store_path):
6.          # path to code repository
7.          self.code_path = code_path
8.          self.store_path = store_path
9.
10.         # path of all python files
11.         self.all_py_files = glob("%s/**/*.py" % str(code_path), recursive=True)
12.
13.         # acquire the trunk of extracted ast nodes, all of the nodes are functions.
14.         self.ast_nodes = self.extract_code_to_astnodes()
```

Under the "Doc2vecDataExtractor" class, the "extract_code_to_astnodes" function generates all the ast notes using the pyan module. All of the extracted nodes are functions. The attribute "ast_nodes" stores the trunk of extracted ast nodes. The class contains an export function that writes the generated data to an output file. Each line in the output file is a function name and its source code separated by a space character.

The code snippet below shows the general pipeline for doc2vec embedding approach.

```
1.  def run_doc2vec(dataset_path:str):
2.
3.      dataset_save_path = prepare_dataset(dataset_path, "doc2vec.txt")
4.
5.      d2v = Doc2Vec(dataset_save_path)
6.      d2v.train_model()
7.      d2v.save_model("d2v_model")
8.      d2v.evaluate()
```

The function prepare_dataset collects the data extracted by "Doc2vecDataExtractor". It then splits the dataset into training and testing sets.

```
class Doc2Vec:
    '''
        doc2vec is a method that learns paragraph and document embeddings.
        It is proposed by Mikilov and Le in 2014.
        This class includes a Gensim implementation of doc2vec.
    '''

    def __init__(self, data_dir):
        self.data_dir = Path(data_dir).resolve()
        self.train_path = data_dir / "train.txt"
        self.test_path = data_dir / "test.txt"

        self.train_corpus = list(self.read_dataset(self.train_path))
        self.test_corpus = list(self.read_dataset(self.test_path, tokens_only=True))

        self.train_tags = list(self.read_keywords(self.train_path))
```

For training the embeddings, we created a class called “Doc2vec”. The attributes “train_corpus” and “test_corpus” is a list of functions and their source code. The “train_corpus” will be fed into the doc2vec model.

```
def train_model(self, vector_size=50, window=2, min_count=2, epochs=100, workers=4):
    self.model = gensim.models.doc2vec.Doc2Vec(vector_size=vector_size, window=window,
                                               min_count=min_count, epochs=epochs,
                                               workers=workers)
    self.model.build_vocab(self.train_corpus)
    self.model.train(self.train_corpus, total_examples=self.model.corpus_count,
                     epochs=self.model.epochs)
```

We used the gensim implementation of doc2vec model to train our embeddings. The “train_model” function initializes the doc2vec model and performs training.

```
def evaluate(self):
    true_positives = 0
    false_positives = 0
    false_negatives = 0
    nr_predictions = 0

    for tag, keywords_doc in self.test_corpus:
        nr_predictions += 1

        inferred_vector = self.model.infer_vector(keywords_doc)
        sims = self.model.docvecs.most_similar([inferred_vector], topn=10)

        inferred_names = [sim[0] for sim in sims if sim[0] in self.train_tags][:1]
        print(tag, inferred_names)
        original_subtokens = self.get_subtokens(tag)

        for inferred_name in inferred_names:
            inferred_subtokens = self.get_subtokens(inferred_name)
            true_positives += sum(1 for subtoken in inferred_subtokens
```

```
20.                 if subtoken in original_subtokens)
21.             false_positives += sum(1 for subtoken in inferred_subtokens
22.                 if subtoken not in original_subtokens)
23.             false_negatives += sum(1 for subtoken in original_subtokens
24.                 if subtoken not in inferred_subtokens)
25.
26.     precision = true_positives / (true_positives + false_positives)
27.     recall = true_positives / (true_positives + false_negatives)
28.     f1 = 2 * precision * recall / (precision + recall)
```

The function “evaluate” in the class can be used to evaluate the performance of the doc2vec model. The “most_similar” function returns functions from the training dataset that are most similar to the input test function. We evaluate the performance of this inference by using sub-token matching; from line 19 to line 24, metrics like true positives, false positives and false negatives are calculated by matching sub-tokens between original label and inferred label. We break function labels down into subtokens by separating it into individual words. For example, “sampleFunctionName” is broken down into the following sub-tokens: sample, function, Name. We then calculate the precision, recall and f1 scores based on the sub-token matching metrics as shown from line 26 to line 28.

2.3.2 Code2vec

Therefore, in addition to the doc2vec model, we also explored the embedding approach that considers the syntactic architecture of source code. In the code2graph pipeline, while acquiring the Lightweight graph, we also store the Abstract Syntax Tree (AST) for each function node. To represent these functions, we exploited the AST stored in each function node that contains static structural information in the form of parent-child node relationships. One famous work that utilizes AST to learn code embeddings is [4]. Using code2vec, we first decompose an AST: G = (V, E), where V is the set of vertices and E is the set of edges, into a collection of context paths. Each context path <e1, p, e2> uses a triple format where e1 and e2 stand for the leaf nodes on G and p is a path that connects e1 and e2. The idea of learning the code embeddings is to model each function as a bag (multiset) of its extracted context paths. In the learning process, code2vec not only acquires the latent representations for leaf nodes and paths in the context paths but also learns how to aggregate the bag of context paths as code vectors with a path-attention layer. The code2vec trains in a supervised manner using the names of functions as the learning objective.

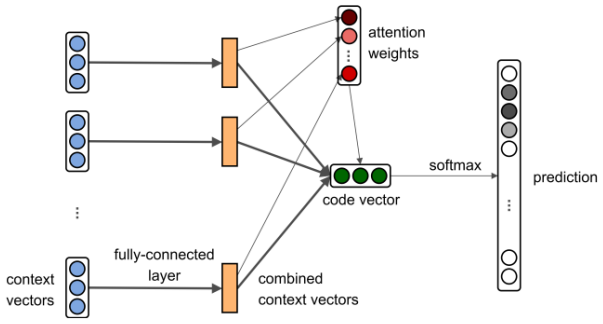

Figure 8. The path-attention neural network architecture used in code2vec.

We created a library[12] for our implementation of code2vec pipeline. Our library performs data extraction and preprocessing, training and evaluation. For data extraction, we created a class called Code2vecDataExtractor as follows:

```
1.  class Code2vecDataExtractor:
2.
3.      def __init__(self, code_path, store_path):
4.          # path to code repository
5.          self.code_path = code_path
6.          self.store_path = store_path
7.
8.          # path of all python files
9.          self.all_py_files = glob("%s/**/*.py" % str(code_path), recursive=True)
10.
11.         # leaves nodes of ast tree for each module
12.         # paths between leaves of ast tree for each module
13.         self.leaves, self.paths = {}, {}
14.
15.         # hyperparameters
16.         self.path_length_upper_bound = 10
17.         self.path_length_lower_bound = 3
18.
19.         # acquire the trunk of extracted ast nodes, all of the nodes are functions.
20.         self.ast_nodes = self.extract_code_to_astnodes()
```

Under the “Code2vecDataExtractor” class, the “extract_code_to_astnodes” perform the same function like the one in the “Doc2vecDataExtractor”. The attributes paths and leaves are dictionaries that store the paths and the leaves of the ast tree generated for each module.

```
1.  def process_all_nodes(self):
2.      self.generate_leaves()
3.      self.generate_paths()
4.
5.      for node_name, node in self.ast_nodes:
6.          print(node_name, ' gets %s leaves' % len(self.leaves[node_name]))
7.          print(node_name, ' gets %s paths' %  len(self.paths[node_name]))
```

The function “process_all_nodes” contains the main routine that extracts the data for the code2vec embedding approach. It calls the function “generate_leaves” on line 2, which acquires all the leaves of a module or function by traversing through its ast subtree. Then the function “generate_paths” on line 3 iterates through all the pairs of leaves and acquires all paths between the pair of leaves. The extracted data is represented as a triple by joining the leaves and paths with tab space character '\t' as follows: "leaf1\tpath\tleaf2". Then all the triples for each function are joined with space character. The "export"

[12] https://github.com/louisccc/code2vec

function in the class can be used to write the extracted data into an output file. Each line in the output file is a function label and its list of triples separated by space character.

The following code snippet shows the general pipeline of the code2vec embedding approach.

```
1.  def run_code2vec(dataset_path:str):
2.
3.      dataset_save_path = prepare_dataset(dataset_path, "code2vec.txt")
4.
5.      trainer = Trainer(dataset_save_path)
6.      trainer.train_model()
7.      trainer.evaluate_model()
```

The function “prepare_dataset” on line 3 collects the data extracted by “Code2vecDataExtractor”. It parses through the dataset and removes functions with the number of triples less than 200. Then it splits the dataset into training and testing sets. Our model requires each function to have exactly 200 triples, so we sample 200 triples from a list of total triples for each function. To ensure that we capture all the information of a function in the training dataset, we resample the triples (total_number_of_triples / 200) times. Next, we assign an identification number (id) to represent each unique entity, path, and label. Then we create dictionaries that map entities, paths, and the labels to their ids and vice versa. Finally, we translate the dataset into ids using the mappings.

```
1.  class Trainer:
2.      ''''' the trainer for code2vec '''
3.      def __init__(self, path):
4.          self.config = Config()
5.
6.          self.reader = PathContextReader(path)
7.          self.reader.read_path_contexts()
8.          self.config.path = path
9.
10.         self.config.num_of_words = len(self.reader.word_count)
11.         self.config.num_of_paths = len(self.reader.path_count)
12.         self.config.num_of_tags  = len(self.reader.target_count)
13.
14.         self.model = code2vec(self.config)
15.         self.optimizer = tf.keras.optimizers.Adam(
16.             learning_rate = self.config.learning_rate)
```

We created the “Trainer” class to train the code embeddings using the code2vec method; the attribute “model” on line 14 holds an instance of code2vec model. The attribute “reader” stores an instance of a class called “PathContextReader”, which loads the training and testing dataset and splits them into batches. We call the “train_model” function to train the model in batches as follows:

```
1.  def train_model(self):
2.
3.      self.train_batch_generator = Generator(self.reader.bags_train, self.config.training
    _batch_size)
```

```
4.
5.      for epoch_idx in tqdm(range(self.config.epoch)):
6.
7.          acc_loss = 0
8.
9.          for batch_idx in range(self.train_batch_generator.number_of_batch):
10.
11.             data, tag = next(self.train_batch_generator)
12.
13.             e1 = data[:,:,0]
14.             p  = data[:,:,1]
15.             e2 = data[:,:,2]
16.             y  = tag
17.
18.             loss = self.train_step(e1, p, e2, y)
19.
20.             acc_loss += loss
21.
22.         if epoch_idx % 5 == 0:
23.             print("Evaluation Set Test:")
24.             self.evaluate_model(training_set=False)
25.             print("Training Set Test:")
26.             self.evaluate_model(training_set=True)
27.             self.save_model(epoch_idx)
28.             self.export_code_embeddings(epoch_idx)
29.
30.         print('epoch[%d] ---Acc Train Loss: %.5f' % (epoch_idx, acc_loss))
```

The function “evaluate_model” on line 24 evaluates the model using the same logic from the “evaluate” function of doc2vec. For each function in the testing dataset, the code2vec model predicts the label names or tags. We use sub-token matching between original label tags and predicted label tags to calculate the precision, recall, and F1 scores of the model.

2.3.3 Pykg2vec

The RDF graphs extracted from both the Computation graph-based approach and the Lightweight approach can be formulated as Knowledge Graphs (KGs), as illustrated in Figure 9. For a given KG, it contains a set of entities E and relations R between entities. For the Computation Graph-Based approach, we formulated the simplified graphs as KGs, where the entities stand for operators and virtual nodes, and the relations stand for various connections between these nodes. For the Lightweight Approach, we formulated the Lightweight graph as KGs where the entities stand for the names of the functions, and the relations stand for the dependencies (calls, or followed by) between function calls. With these formulations, we utilized Knowledge Graph Embedding (KGE) algorithms to learn embeddings for nodes and edges. The set of facts D+ in the KG (extracted using RDF N-Triple form in code2graph) are represented in the form of triples (h, r, t), where h, t ∈ E are referred to as the head, and the tail functions and r ∈ R is referred to as the relationship. Take one triple in the Lightweight graph as an example. Triple “<Dense_1> \t <followedBy> \t <Dense_2>” indicates that the function invocation Dense_2 is executed after Dense_1. In addition to D+, KGE models usually need a set of unseen or negative training samples under the Open World Assumption (OWA) in training. Then, a scoring function $f_r(h, r)$ calculates the loss and gradient for updating the embeddings. The

evaluation of pykg2vec KGE methods lies in their capability of predicting the missing functions in negative triples (?, r, t) or (h, r, ?) or predicting whether an unseen fact is true or not. The evaluation metrics include the rank of the answer in the predicted list (mean rank), and the ratio of answers ranked top-k in the list (hit-k ratio). Our library: pykg2vec[13] includes a collection of KGE models and tools for evaluation. In code2graph, we utilized pykg2vec to evaluate both computation graphs and lightweight graphs.

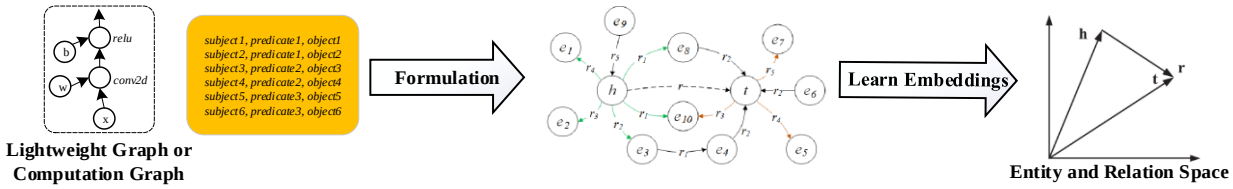

Figure 9. The pipeline of applying Knowledge Graph Embedding (KGE) Methods.

To preprocess the dataset for training KGE models, we used the Computation Graph-based approach and the Lightweight approach. The implementation details of these two data extraction approaches was described in the previous technical report. We then used the processed dataset to train KGE models; to do this we created a script that uses the pykg2vec library. The main pipeline of the script is as follows:

```
1.  def run_pykg2vec():
2.      # getting the customized configurations from the command-line arguments.
3.      args = PyKG2VecArgParser().get_args(sys.argv[1:])
4.      args.dataset_path = preprocess(args.triples_path, args.dataset_name)
5.
6.      # Preparing data and cache the data for later usage
7.      knowledge_graph = KnowledgeGraph(dataset=args.dataset_name,
8.          negative_sample=args.sampling, custom_dataset_path=args.dataset_path)
9.      knowledge_graph.prepare_data()
10.
11.     # Extracting the corresponding model config and definition from Importer().
12.     config_def, model_def = Importer().import_model_config(args.model_name.lower())
13.     config = config_def(args=args)
14.     model = model_def(config)
15.
16.     # Create, Compile and Train the model. While training, several evaluation will be
        performed.
17.     trainer = Trainer(model=model, debug=args.debug)
18.     trainer.build_model()
19.     trainer.train_model()
```

First, we load our dataset, which is a collection of RDF graphs generated by Lightweight or Computional Graph-based approach in triples format, as shown in line 7-9. Then we define the model and its

[13] https://github.com/Sujit-O/pykg2vec

hyperparameters using the command-line arguments in line 12-14. Finally, we build and train the model in line 17-19.

3 Results

Dataset Statistics: At the beginning of the DCC project, we extracted 10 Computation graphs utilizing the 100-dataset provided by Siemens. We call this dataset 10_Comp. We then configured the PWCScraper to crawl the new papers with repositories on paperswithcode.com continuously. At the end of the DCC Phase, we have crawled around 500 papers from the paperswithcode.com. Out of these 500 papers, 137 papers have a TensorFlow based repository, and the Lightweight Approach can extract 60 graphs from them can. We call this 60_Lightweight. At the end of the DCC Phase 2, we enlarge the dataset to be 2121 repositories, and the Lightweight Approach can successfully process 197 out of them. We call this dataset 197_Lightweight. The datasets we created for evaluating doc2vec and code2vec have 244 repositories processed by Doc2vecDataExtractor and Code2vecDataExtractor. Each sample in the lightweight dataset represents a function call. Each sample in doc2vec and code2vec dataset represents a function. In Lightweight datasets (60_Lightweight and 197_Lightweight), the entities represent functions or properties of functions, and the relations indicate the dependencies between functions or indicate the containing relationship between a function and an attribute. In the code2vec dataset, the entities stand for leaves on an AST, and the relations represent the context paths between two leaves.

Metadata Information	60_Lightweight	197_Lightweight	doc2vec dataset	Code2vec dataset
Total # of Repos	117	2121	2121	2121
Total # of handled repos	91	197	244	244
Total Training Samples	85851	169197	-	98426
Total Testing Samples	26530	52236	-	1627
Total validation Samples	20265	39744	-	0
Total Entities	10154	18413	-	73948
Total Relations	284	384	-	1729086

Table 1. The Data Statistics of 60_Lightweight, 197_Lightweight, doc2vec, and code2vec datasets.

Train on Pykg2vec: During Milestone 8 of the DCC project, we have improved the pykg2vec so that it includes state-of-the-art KGE models like Complex-N3, ConvE, and TuckER that can capture more sophisticated relationships between entities and relations. In the DCC repository, script_training_pykg2vec.py can trigger the training and evaluation using pykg2vec as the backend. The evaluating task in pykg2vec is the link prediction of the missing parts in the RDF triple (?, r, t) or (h, r, ?). As for metrics, mean rank is the averaged number over the positions of the missing h (head rank) or t (tail rank) on the list of predicted candidates. The mean rank reciprocal is the average of each rank's reciprocal. Hit-k indicates the ratio of the link predictions with a rank lower than k. We conduct the evaluation tasks on the testing set of each dataset. Without extensive hyperparameter tuning, we ran the KGE models using the corresponding golden settings used in training benchmarks. From Table 2, we notice that the difference of metrics is subtle between training on a smaller dataset and training on a larger dataset (60_Lightweight v.s. 197_Lightweight), indicating the size of both datasets should be enough for training KGE methods. The KGE model that achieves the best performance is ConvE, which is a convolution-based approach with

0.721 and 0.724 mean_rank_reciprocal for both datasets, respectively. We can interpret the number 0.7 in mean_rank_reciprocal as that the learned models can correctly predict the missing entities over 70 percent of the link predictions. We can also observe a similar trend in the Hit-1 Ratio. Simple KGE models, such as TransE and TransH, perform worse in the link prediction task compared to state-of-the-arts KGE models such as ComplexN3, ConveE, and TuckER. The results indicate that the learned embeddings for entities and relations preserve the meaningful information for link prediction task, which we can utilize to benefit other inference tasks related to these knowledge graphs (node or graph classification).

Models	Dataset	mean_rank_reciprocal	mean_rank	Hit-1 Ratio	Hit-3 Ratio	Hit-5 Ratio	Hit-10 Ratio
TransE	60-lightweight	0.463	338.743	0.321	0.571	0.638	0.701
TransH	60-lightweight	0.386	327.829	0.211	0.513	0.601	0.686
Analogy	60-lightweight	0.693	367.248	0.650	0.724	0.736	0.753
ComplexN3	60-lightweight	0.692	421.363	0.645	0.725	0.737	0.755
ConvE	60-lightweight	0.721	731.833	0.705	0.725	0.736	0.753
TuckER	60-lightweight	0.616	375.875	0.570	0.637	0.664	0.702
TransE	197-lightweight	0.425	631.164	0.299	0.510	0.582	0.657
TransH	197-lightweight	0.392	585.196	0.257	0.470	0.553	0.645
Analogy	197-lightweight	0.682	680.822	0.632	0.722	0.736	0.752
ComplexN3	197-lightweight	0.678	748.709	0.620	0.725	0.738	0.755
ConvE	197-lightweight	0.724	1349.968	0.710	0.727	0.736	0.752
TuckER	197-lightweight	0.613	727.234	0.575	0.628	0.653	0.688

Table 2. The results of running KGE methods using pykg2vec on 60_Lightweight, 197_Lightweight datasets.

On the other hand, Figure 10 shows the T-SNE visualization of the learned embeddings trained with TransE model. The figure demonstrates the distribution of all the entities including TensorFlow API, Userdefined API, and modules invocations. The result demonstrates the clustering of entities where userdefined functions are staying together on the left side of the embedding space, indicating that the training was meaningful in that the nodes with similar functions are closer on the embedding space.

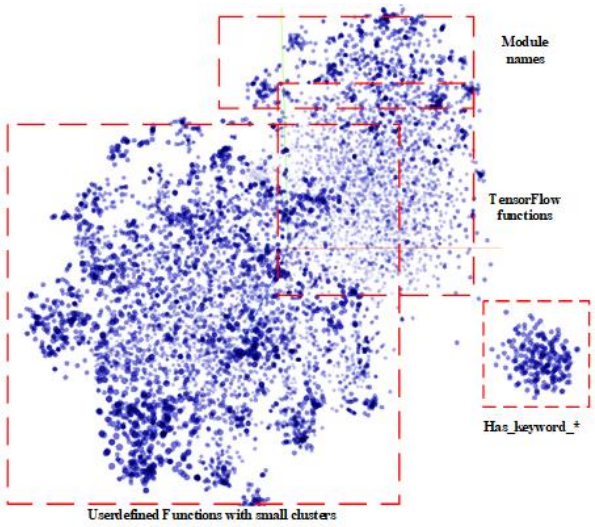

Figure 10. The visualization of all the function embeddings trained with the TransE model.

Train on Code2vec and Doc2vec: During Milestone 9 of the DCC project, to evaluate similarity, we have experimented with doc2vec and code2vec models that can learn the code vectors (we first target the function level). We adopted the measure used in code2vec and [5] which evaluate the quality of function name predictions by matching the subtokens of the inferred labels and the true labels. If a subtoken exists in both inferred labels and true labels, we count it as a true positive. If a subtoken exists in the inferred label but not in the true label, we count it as a false positive. If a subtoken exists in the true label but not in the inferred label, we then count it as a false negative. Using these numbers, we calculate the Precision and Recall and F1 as the metrics for evaluation. For code2vec, we have also appended the mean rank and the mean rank reciprocal numbers to evaluate the ranks of prediction. The results are shown in Table 3. From the table, we can see code2vec performs slightly better than doc2vec, as it considers the syntactic information in the source code. However, the performance difference is not that significant as the code2vec model is essentially a data-hungry network architecture that requires millions of training datapoints to improve results. Also, the mean rank reciprocal for the code2vec model is around 0.45, indicating that 45 percent of the predictions are exact. In the future, with these results, we can increase the resolution so that we can then compare repositories.

Models	Precision	Recall	F1	Mean Rank	Mean Rank Reciprocal
doc2vec	0.524	0.533	0.528	-	-
code2vec	0.570	0.557	0.563	1694.01	0.453

Table 3. The results of running code2vec and doc2vec on doc2vec and code2vec datasets (created from 197_Lightweight).

4 Discussion & Conclusion

After completing 9 Milestones in the DCC project, we have developed two main graph extraction approaches and also graph embedding approaches. Throughout the project, the number of repositories that code2graph graph extraction pipeline can process is one potential direction for further improvement. In recent, we notice a trend in the 197_Lightweight dataset that over 50 percent of these 2000 repositories use

PyTorch[14] as its programming framework. With this, we can acquire a larger dataset if we extend the Lightweight Approach to handle PyTorch repositories.

In the DCC project, we have applied graph embedding approaches to learning the representations of functions in code repositories. The next step is to increase the abstraction level to repositories so that we can have similarity measure and novelty measure in embedding space. For doc2vec and code2vec models, we could simply expand the bag of context paths or the bag of text pieces so that it covers the whole repository, but this requires a larger dataset to train the models. As Figure 11 illustrates, we can alternatively train an auto-encoder like architecture to acquire the representations for each knowledge graph. A similar option involves using a Graph Neural Network (GNN) with a graph-pooling layer. Once we learn the embeddings for KGs, we can perform similarity measurement by calculating the distance between repositories on the embedding space.

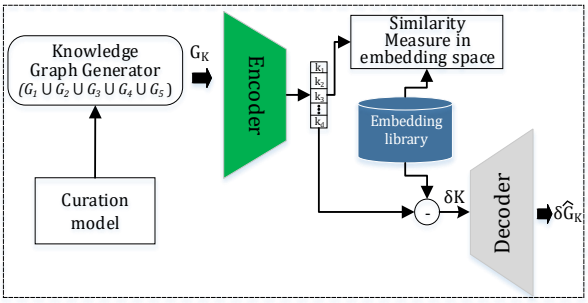

Figure 11: The architecture of learning KG embeddings using auto-encoder and similarity comparison on repositories.

5 References

[1] S.-Y. Yu, A. S. Aksakal, S. R. Chhetri and M. A. A. Faruque, "Deep Code Curator – Technical Report on Code2Graph," 2019.

[2] T. Mikolov, K. Chen, G. Corrado and J. Dean, "Efficient Estimation of Word Representations in Vector Space," 16 1 2013.

[3] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*, 2014.

[4] M. Andrychowicz and K. Kurach, "Learning Efficient Algorithms with Hierarchical Attentive Memory," 9 2 2016.

[14] https://pytorch.org/

[5] U. Alon, M. Zilberstein, O. Levy and E. Yahav, "A general path-based representation for predicting program properties," *ACM SIGPLAN Notices*, vol. 53, p. 404–419, 2018.

[6] M. Cvitkovic, B. Singh and A. Anandkumar, "Deep learning on code with an unbounded vocabulary," in *Machine Learning for Programming (ML4P) Workshop at Federated Logic Conference (FLoC)*, 2018.

[7] T. Ben-Nun, A. S. Jakobovits and T. Hoefler, "Neural Code Comprehension: A Learnable Representation of Code Semantics," 19 6 2018.

[8] U. Alon, S. Brody, O. Levy and E. Yahav, "code2seq: Generating Sequences from Structured Representations of Code," 4 8 2018.

[9] M. Allamanis, M. Brockschmidt and M. Khademi, "Learning to Represent Programs with Graphs," 1 11 2017.

[10] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[11] M. Allamanis, E. T. Barr, C. Bird and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[12] M. Allamanis, E. T. Barr, P. Devanbu and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, p. 1–37, 2018.