Cross-language clone detection by learning over abstract syntax trees

1st Daniel Perez*
Imperial College London
London, United Kingdom
daniel.perez@imperial.ac.uk

2nd Shigeru Chiba
The University of Tokyo
Tokyo, Japan
chiba@acm.org

***Abstract*—Clone detection across programs written in the same programming language has been studied extensively in the literature. On the contrary, the task of detecting clones across multiple programming languages has not been studied as much, and approaches based on comparison cannot be directly applied. In this paper, we present a clone detection method based on semi-supervised machine learning designed to detect clones across programming languages with similar syntax. Our method uses an unsupervised learning approach to learn token-level vector representations and an LSTM-based neural network to predict whether two code fragments are clones. To train our network, we present a cross-language code clone dataset — which is to the best of our knowledge the first of its kind — containing around 45,000 code fragments written in Java and Python. We evaluate our approach on the dataset we created and show that our method gives promising results when detecting similarities between code fragments written in Java and Python.**

***Index Terms*—clone detection, machine learning, source code representation**

I. Introduction

Code clones are fragments of code, in single or multiple programs, which are similar to each other. Code duplication can decrease the maintainability of a program, as it becomes necessary to fix an error in all the places where the code was duplicated. Detecting these code clones is a difficult task and has been extensively researched in the literature [1]–[3]. Some systems focus on finding clones inside a single project, while other systems try to detect clones in larger ecosystems [3], [4]. While there is a very large number of tools that have been developed for the task of clone detection, most of these have been developed to detect clones in programs written in the same programming language, and the task of detecting code clones for programs written in different languages has not been studied as much in the literature.

Although systems written in multiple programming languages have always existed, they have now become the rule rather than the exception [5]. A common case of systems written in multiple programming languages is systems following the microservice architecture [6], which have gained a large adoption as a scalable architecture for web applications. While this architecture by itself does not imply using different programming languages, developers often use the language which is best suited for a task [7], resulting in the use of multiple languages. Many large companies such as Facebook [8], Uber [9] or Netflix [7], followed by many others [10], use such an architecture and have code bases written in a variety of programming languages.

In this paper, we present a semi-supervised machine learning based system capable of finding code clones across programming languages with similar syntax. We make the following contributions.

- We present a cross-language clone detection method and provide a prototype implementation supporting clone detection across Java and Python
- We create a cross-language code clones dataset containing around 45,000 files written in Java and Python with annotations about which of the files are code clones

We make the source code of our code clone detection system as well as all the datasets we created for the experiments publicly available[1].

II. Background

Previous works have shown that code clone detection can help to refactor and improve the maintainability of large code bases [11]. However, previous works focus almost only on clone detection within a single programming language.

In systems following the microservice architecture, which is widely adopted for web applications, services are often written by different teams. This makes it hard for developers to track code and functionality duplication across different services. Furthermore, as these services can be written in different programming languages, current clone detection approaches are not applicable to detect duplication automatically. However, there are some classes of code clones that may negatively affect the maintainability of a system and which code clone detection tools could help to prevent.

For example, a common reason for the appearance of clones in a microservice context is when a service breaks the single-responsibility principle [12]. For example, a service which is responsible for managing the user posts, may at some point implement some authorization logic. Then, another service responsible for managing comments will also implement a

*Work done while at The University of Tokyo

[1] https://www.csg.ci.i.u-tokyo.ac.jp/projects/clone/

```
def group_posts(posts):
    res = {}
    for post in posts:
        bucket = res.setdefault(post.owner, [])
        bucket.append(post)
    return res
```

Listing 1. group_posts function in Python

```
public Map<String, List<Post>> groupPosts(
    List<Post> posts) {
  Map<String, List<Post>> grouped =
    new HashMap<>();
  for (Post post: posts) {
    if (!grouped.containsKey(post.getOwner())) {
      grouped.put(post.getOwner(),
                  new ArrayList<Post>());
    }
    grouped.get(post.getOwner()).add(post);
  }
  return grouped;
}
```

Listing 2. groupPosts method in Java

similar authorization logic, resulting in duplicated functionality.

Although all classes of cross-language clones are not trivial to refactor, there are some patterns that can be used to reduce such duplication. The system can be re-designed to extract functionality into a new micro-service which other services can reuse. Another common approach is to add the functionality to a service that is already consumed by the services with the duplicated code.

We show a simple example of a duplicated functionality implemented in Python in Listing 1 and Java in Listing 2. Although the two code snippets are written in different languages, we see a clear correspondence between their control structures. Their Abstract Syntax Trees (AST) also present some similarities. For example, both the body of the Python function and the body of the Java method contain three children — an assignment, a `for` loop and a `return` statement. We show the relevant part of each AST in figures 1 and 2.

We see the possibility to detect AST resemblance but the existing approaches are not effective for this use case. Both snippets only share very few tokens in common, making token based clone detection methods such as [4] ineffective. The AST of the fragments, although sharing some similarities, are too different for approaches such as [13] or [14] which directly compare ASTs to be applied.

The kind of AST resemblance described above does not fit well the commonly used taxonomy of code clones [1]. Type III clones are often assumed to come from copied fragments while type IV clones do not assume anything on the structure of the clone ASTs. The two fragments above could potentially be seen as weakly Type III using the definition given in [15] — syntactically similar with less than 50% similarity at the statement level — although the syntactic similarity is not straightforward due to the cross-language nature of the clones.

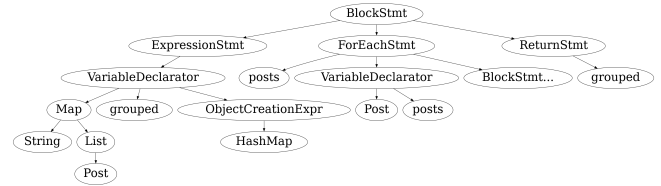

Fig. 1. Java groupPosts method AST

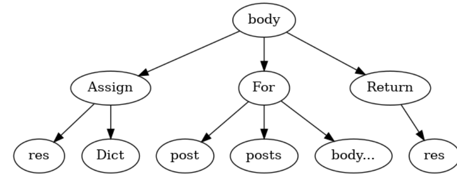

Fig. 2. Python group_posts function AST

We discuss issues with the current taxonomy further in IV-C.

One of the main challenges in detecting such clones is to map AST subtrees between programming languages. In the above example, the assignment in Python is a simple `Assign` node with two children while the Java assignment is much more complex, starting with an `ExpressionStmt` and containing nodes to declare a generic `HashMap` as well as nodes to instantiate a new object. Furthermore, some idioms can differ even more between languages. For example, the `setdefault` at line 4 of the Python example is replaced by an `if` statement in Java. Manually creating and maintaining such rules for multiple languages seems almost impossible and we, therefore, want a way to learn such mappings automatically.

III. Proposal

In this work, we propose a semi-supervised machine learning based system which is capable of detecting code clones across programming languages with similar syntax. A key component of this system is our token-level vectors generation algorithm, tree-based skip-gram, which generates a semantically meaningful mapping from a token to a point in a vector space. In the context of cross-language clone detection, assigning a meaningful vector representation to each token is particularly important as it makes it easier for the rest of the model to map subtrees — such as the `HashMap` constructor and the dictionary literal in the previous example — across languages.

We will first give a general overview of our system, then give details about our tree-based skip-gram algorithm and finally describe our clone detection model.

A. System overview

Our system trains a clone detection model and uses it to discover clones. It is mostly composed of a token-level vector generation step which is described in depth in III-B, a training step in which we use the cross-language clones dataset we created to train the model, and finally, a clone detection step, both described in III-C.

During the token-level vector generation step, the system generates a fixed-size vocabulary for each target programming language, as well as a vector representation of every token in the vocabulary. This step is unsupervised and simply requires a large amount of code for each targeted programming language. Details about vocabulary and token-level vector generation using our tree-based skip-gram algorithm are given in III-B.

Once the token-level vectors are generated, the next step is to train our clone detection model. The model being supervised, this step requires to have an annotated dataset containing information about code clones written in the targeted programming languages. The model uses the token-level vectors, computed in the previous step, to transform each node in the ASTs into vectors. It simultaneously learns to encode a whole AST into a large vector, and to classify whether the two vectors are clones or not, using the labels in the dataset provided. Details about the model and the training process are given in III-C.

The last step is the actual code clone detection. To perform clone detection, our system uses the vocabulary and token-level vectors, as well as the clone detection model generated in the previous steps. Using these, the system first vectorizes all the code fragments for which clone detection should be performed and then runs the classifier trained in the previous step on each pair of fragments to search for clones. This step is detailed further in III-D2.

B. Token-level vectors generation

Our token-level vectors generation algorithm, tree-based skip-gram, is based on the skip-gram algorithm [16] but uses the structure of the AST to compute a vector representation of each token in a target programming language. While the skip-gram algorithm treats its input as a sequence and generates the context of a particular target using the tokens around it, tree-based skip-gram uses the tree structure to generate the context for a particular target. Tree-based skip-gram is a base technique which can be used to help to find particular shapes of subtrees or compare subtrees in multiple ASTs.

The process for generating token-level vectors using tree-based skip-gram for a target programming language $\mathcal{L}$ is the following.

1) Collect a large amount of source code written in $\mathcal{L}$
2) Parse the code to generate AST representation
3) Generate a vocabulary from the collected source code
4) Generate target and context pairs using the parsed ASTs
5) Train a skip-gram model with the generated target and context pairs

We show an overview of the token-level vectors generation process in Figure 3. We will describe the algorithms to generate the vocabulary and generate the data to train the skip-gram model, and give details about how we train the model. We will provide more details about the source code collection in Section IV.

1) Vocabulary generation: To be able to learn token-level vectors for the target programming language, we first generate a finite set of tokens: the vocabulary. Each token has a type,

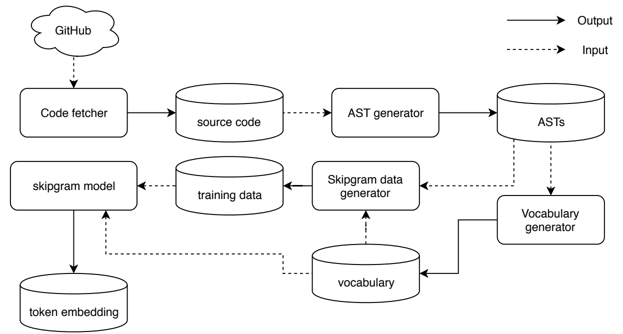

Fig. 3. Token-level vectors generation overview

Algorithm 1 Vocabulary generation algorithm

```
 1: function GENERATEVOCABULARY(files, includeValues, maxSize)
 2:   tokensCount ← empty map
 3:   for file in files do
 4:     ast ← generate_ast(file)
 5:     for token in ast do
 6:       if (token.type, null) ∉ tokensCount then
 7:         tokensCount[(token.type, null)] ← 0
 8:       Increment tokensCount[(token.type, null)]
 9:       if includeValues ∧ token.value ≠ null then
10:         if (token.type, token.value) ∉ tokensCount then
11:           tokensCount[(token.type, token.value)] ← 0
12:         Increment tokensCount[(token.type, token.value)]
13:   tokensCount ← reverse_sort(tokensCount)
14:   vocabulary ← first maxSize keys of tokensCount
15:   return vocabulary
```

for example, `ForStmt` or `NameExpr` and may have a value which is usually an identifier name and is often application specific. While the number of token types is finite, the number of token values is infinite — it could be any user-defined identifier. This means that we must put a threshold on the size of the vocabulary and when using our vocabulary, it will not contain all possible tokens. In most NLP applications, tokens not present in the vocabulary are replaced by a unique "unknown" token. However, in the context of programming languages, the probability of running into unknown values is much higher than in NLP and we, therefore, want to avoid using a generic "unknown" token. Instead, we choose to keep the type information of the token and only replace the token value by `null` when no token with the same type and value is found in the vocabulary. This allows us to at least keep some semantic information about the token so that, for example, a string literal and an identifier can be distinguished even if their value was not found in the dictionary. We show how we generate the vocabulary V in Algorithm 1. In order to be able to fall back to the type as described above, when generating V, we want the following property to hold.

Property 1. *Given A the set of token types in programming language $\mathcal{L}$ and P the set of programs used to generate vocabulary V, if $|A| \leq |V|$ then*

$$\forall a \in A, a \in P \rightarrow (a, \texttt{null}) \in V$$

However, property 1 might not hold if `reverse_sort`

function, used at line 13 of Algorithm 1, is defined to only order with respect to the number of appearances of a token. To overcome this issue, given v_t the value of a token and c_t its number of appearances in the vocabulary, we use the following order $\leq$ on the tokens to perform the sort.

$$t_1 \leq t_2 = \begin{cases} \texttt{false} & \text{if } v_{t_1} = \texttt{null} \wedge v_{t_2} \neq \texttt{null} \\ \texttt{true} & \text{if } v_{t_1} \neq \texttt{null} \wedge v_{t_2} = \texttt{null} \\ c_{t_1} \leq c_{t_2} & \text{otherwise} \end{cases} \quad (1)$$

Using the order defined in equation 1, the vocabulary produced by Algorithm 1 respects property 1.

Proof. If a token of type a is included in P, then an entry $(a, \texttt{null})$ will be created. As the order described above ensures that all entries where the value is `null` are greater than other values, a reverse sort will ensure that these will appear before other tokens. Therefore, if `maxSize` is equal or greater to the number of type token created, they will all be included in the vocabulary. □

As the vocabulary is typically generated from a large corpus, we can almost be sure that all token types of programming language $\mathcal{L}$ will be included in the set of programs P. By putting this together with property 1, we can conclude that for all token types in $\mathcal{L}$, a pair $(a, \texttt{null})$ will be included in the vocabulary. Using this property, we can define our vocabulary lookup function very easily. If the pair of the token type and its value is in the vocabulary, we return its index. Otherwise, we return the index of the pair defined by the token type and `null` — $(a, \texttt{null})$.

2) Skip-gram data generation: After generating the vocabulary, we generate data to train a skip-gram model. In the context of natural language processing, the input is usually considered as a sequence, and the context of a particular word is the words before and after this word in the sequence of words used for training. Furthermore, the distance between the word and its context is normally parameterized by a single window size hyperparameter. In our tree-based skip-gram algorithm we take advantage of the topological information contained by the AST instead of working on a simple sequence of tokens. Therefore, we need to define the context of a token differently than for a sequence.

In the context of an AST, a node is directly connected to its parent and its children. Hence, we can define parents and children to be the context of a node. Depending on the use case, the siblings of a node could also be viewed as viable candidates for its context. A single window size parameter could be used to control how deep upward and downward should the context of a node be. However, although a node will only have a single parent, yet it can have any number of children. Therefore, having a window size of 3 for the ancestors would only generate 3 nodes in the context, but if every descendant of a node had 5 children, a window size of 3 would generate $5^3 = 125$ nodes in the context. This would probably generate more noise than signal when trying to train the model. Therefore, we use two different parameters

Algorithm 2 Data generation for skip-gram model

```
1: function GenerateSkipgramData(files, vocabulary, params)
2:   skipgramData ← {}
3:   for file in files do
4:     ast ← GenerateAST(file)
5:     for node in ast do
6:       nodeIndex ← LookupTokenIndex(node)
7:       contextNodes ← GenerateContext(node, params)
8:       for contextNode in contextNodes do
9:         contextIndex ← LookupTokenIndex(contextNode)
10:        skipgramData.add((nodeIndex, contextIndex))
11:  return skipgramData
```

Algorithm 3 Context generation for an AST node

```
1: function GenerateContext(node, params)
2:   contextNodes ← FindDescendants(node, params.descendantWS,
     0)
3:   parent ← node.parent
4:   n ← 0
5:   while parent is defined ∧ n < params.ancestorWS do
6:     contextNodes.add(parent)
7:     parent ← parent.parent
8:     n ← n + 1
9:   if params.includeSiblings ∧ node.parent is defined then
10:    for sibling in node.parent.children do
11:      contextNodes.add(sibling)
12:  return contextNodes
```

to control the window size of the ancestors and the window size of the descendant when generating the data to train our skip-gram model. When we do include siblings in the context, we currently use the direct siblings of the nodes and not the siblings of the ancestors, although this could also be another parameter of the algorithm. In algorithms 2, 3 and 4, we describe the process we use to generate the data to train a skip-gram model.

Algorithm 2 takes as input a list of files written in the programming language for which we want to generate token-level vectors, the vocabulary extracted for this programming language and the parameters described above. It loops over all the nodes in the file, uses algorithms 3 and 4 to find all nodes in the context of the current node, and returns a list of pair of indexes where each pair represent a target node and a node in its context. Algorithm 3 takes as input a node and the parameters described above and returns the set of nodes in the context of the given node. It first uses Algorithm 4 to find all the descendants of the node in the window given by the passed parameters, then finds all the ancestors in the given window and finally adds the siblings to the set of results if necessary. Algorithm 4 takes a node, the maximum depth up to which descendants should be populated and the current depth — which will initially be set to 0 — and returns the set of descendants up to the passed maximum depth for the node. It first adds all the children of the current node to the set of descendants, then recurses through all the children until the current depth is equal to the maximum depth for which to generate descendants.

3) Training the skip-gram model: Once the data is generated using algorithms 2, 3 and 4, the last step needed to generate token-level vectors is to actually train a skip-gram

Algorithm 4 Find descendants for a node until given depth

```
1: function FindDescendants(node, maxDepth, depth)
2:   if depth ≥ maxDepth then
3:     return {}
4:   result ← node.children
5:   for child in node.children do
6:     descendants ← FindDescendants(child, maxDepth, depth+1)
7:     result ← result ∪ descendants
8:   return children
```

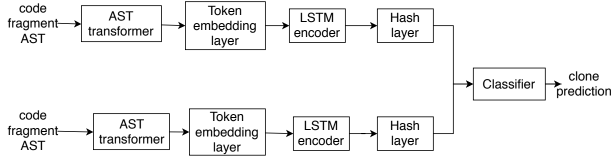

Fig. 4. Clone detection model overview

model using the generated data. Algorithm 2 generates pairs of indexes which can be directly fed to a neural network, and therefore, there is no need for further pre-processing. To train the model, the vocabulary used is the same as the one used to generate the skip-gram data and the size of the vectors is a hyperparameter of the model. The model is trained using the negative sampling objective as given in [16].

C. Clone detection model

Our clone detection model is based on the Siamese architecture [17], which has been popularized in face recognition tasks [18] and has also been used for many Natural Language Processing tasks such as sentence similarity [19]. A major difference between our network and usual Siamese networks is that Siamese networks take their inputs from the same domain — for example, two images or two English sentences. On the other hand, our model takes inputs from different domains: code fragments written in different programming languages. Therefore, unlike regular Siamese networks, we do not share the weights used to encode the inputs. To allow our model to learn efficiently even without sharing weights, we use the token-level vectors precomputed using our tree-based skip-gram algorithm presented in III-B. We show an overview of our model in Figure 4.

Our model is composed of two encoders, which transform each AST into a single vector, and a classifier which outputs a similarity score between the two encoded ASTs. In our implementation, we use the following components.

- AST transformer — Transforms an AST into a vector where each element of the vector is the index of the AST node in the vocabulary. The AST is linearized by ordering its nodes in depth-first order.
- Token embedding layer — Maps each index to its vector representation computed using our tree-based skip-gram model trained by our token-level vectors generation. This is the most unique component proposed in this paper.
- LSTM (Long short-term memory [20]) encoder — Transforms the AST matrix (number of tokens $\times$ vector representation dimension) into a single vector in a high-dimensional space. We use a stacked bidirectional LSTM [21].
- Hash layer — Reduces the dimension of the AST vector outputted by the LSTM. We use a linear layer with no activation function, which weights are trained with the rest of the model.
- Classifier — We use a feed-forward neural network with a sigmoid output layer and therefore get a similarity score between 0 and 1.

We use a binary cross entropy loss to train our model. We give more details about how we choose the code fragments pairs used as input in III-D

The clone detection step uses the trained model. Code fragments are first encoded into vectors, using the encoder part of the model shown in Figure 4. The similarity between the encoded vectors is then computed using the classifier. A pair of code fragments is considered to be a clone if its similarity score is above 0.5. The closer to 1, the most likely it is to be a clone.

D. Implementation details

Our system uses several implementation techniques to improve its speed and precision. We present the most important ones here.

1) Negative clone samples selection: When training our model, we feed it with pairs of code fragments. To create these pairs, we first select a code fragment, which we call the anchor fragment, and then select a positive and n negative samples, where n is a hyperparameter. The number of available clones is relatively small, so to select the positive sample, we randomly select a code fragment in the set of code clones of the current anchor fragment. However, randomly selecting the negative code fragment would likely result in feeding the model with two very different code fragments and the model would therefore only learn to distinguish between code fragments which are very different. To allow the model to distinguish between code fragments which are more similar, we select negative samples that are currently hard for the model to distinguish. This is close to what is done for face recognition in DeepFace [22]. Ideally, we would like to find the code fragments for which the model is the most mistaken — where the predictions between the negative samples and the anchor are the closest to 1. However, this would require to run all the code fragments through our model for each anchor fragment, which is not realistic, performance wise. To work around this, we randomly select m candidate fragments for each anchor, where m is a hyperparameter of the model, usually set to 8 or 16 in our experiments, and run our model on all the candidates. We then sort the candidates using the similarity scores outputted by our model and select the n candidates with the highest similarity score — the n candidates on which the model is the most mistaken — as negative samples for the current anchor.

2) Pre-computing AST vectors: During training, our model takes two code fragments as input and emits a similarity score

between the two code fragments. The model is designed to take pairs of code fragments, this means that to detect clones in n fragments, we need to run the model on all combinations of fragments, resulting in $\mathcal{O}\left(n^2\right)$ runs. Running the whole model, especially the LSTM, is an expensive task and therefore running it $\mathcal{O}\left(n^2\right)$ times would scale very poorly. To work around this issue, we first precompute the output vectors for all the code fragments and therefore only run our LSTM n times. When checking if two code fragments are clones or not, we only need to get the precomputed vector representation of the two fragments and run them through our classifier. Although we still need to run our classifier $\mathcal{O}\left(n^2\right)$ times, it is an order of magnitude cheaper than running the whole model.

IV. Experiments and results

In our experiments, we answer the two following research questions.

- RQ1 Can our system learn similarities between programs written in Java and Python?
- RQ2 Can the vector representation generated by our tree-based skip-gram algorithm improve the ability to learn similarities?

A. Dataset

1) Token-level vectors generation dataset: Tree-based skip-gram being an unsupervised algorithm, to train the model for a particular programming language, the only thing we need is a large quantity of code written in this given language. The code should also be as much as possible diverse so we can generate a representative vocabulary. For Java, we chose to use all the projects written in Java and belonging to The Apache Software Foundation[2] which contain a very wide variety of projects. For Python, as we could not find any organization with a sufficient amount of source code, we chose projects on GitHub which fulfilled the following conditions.

- Size between 100KB and 100MB
- Non-viral license (e.g. MIT, BSD) — to avoid copyright issues when distributing the dataset
- Not forks

We ordered the results by the number of stars as a proxy of the popularity of the project and kept all the files which contained more than 10 tokens and less than 10000 tokens. Although our AST generation tool supports both Python 2 and Python 3, as the produced AST may vary slightly, we decided to use only Python 3 for this experiment. We show some metrics of our dataset in Table I. Although the number of projects is vastly larger for Java, results for both languages were satisfying enough, and thus we did not try to collect more data for Python.

TABLE I
Token-level vectors generation dataset metrics

	Java	Python
Projects count	1,027	879
Files count	476,685	131,506
Lines count	80,367,840	55,796,594
Tokens count	301,930,231	89,757,436

2) Code clones dataset: As our clone detection system is supervised, we need a labeled dataset to be able to train it. In particular, the dataset needs to fulfill the following properties.

1) The dataset should contain code fragments written in at least 2 programming languages
2) Information on whether two code fragments are clones or not should be available

To the best of our knowledge, no dataset currently available fulfills the necessary properties for our experiments and therefore, we created our own dataset.

We found that competitive programming websites mostly fulfill the above properties. The solution to a single problem is implemented by a large number of persons in many different languages. Furthermore, multiple solutions to a problem are always implemented by different users, which makes our dataset closer to the motivating example we presented in Section II. All the solutions to a single problem must implement exactly the same functionality, therefore, we are assured that all source codes implementing a solution to the same problem are type IV code clones. Multiple solutions may implement the same problem using different algorithms making two code fragments marked as clones not having any syntactical similarity. However, the easier the problem is, the higher the probability of code fragments implementing the solution to the same problem has to be very similar to each other, and to therefore be closer to type III clones.

To create the dataset, we used code from a famous competitive programming website[3]. The website we used has two types of contests, regular contests and beginner contests, where beginner contests contain mostly straightforward problems. To increase the probability that the implemented solutions use the same algorithm, and therefore have some syntactical similarities, we used only the code from the beginner contests of the website, which usually have a straightforward solution. As our implementation currently only supports Java and Python, we fetched data for these two programming languages. We restricted the data only to programs that were accepted by the website judging system — meaning that the programs actually implemented the solution to the given problem — in order to reduce noise. We collected code for a total of 576 different problems and give some metrics about the dataset in Table II.

In our experiments, we use each file as a single input to our model. In our dataset, the number of lines per file is of 46 for Java programs and 13 for Python programs. Although this is above the usual size of a single function or method in real-world programs [23], it is still relatively close.

B. Experiments and Results

We performed three different experiments to evaluate our prototype. First, we used our token-level vector generation

[2]http://www.apache.org/

[3]https://atcoder.jp

TABLE II
CLONE DETECTION DATASET METRICS

	Java	Python
Number of problems	576	
Avg. solutions / problem	36	41
Files count	20,828	23,792
Avg. lines / file	46	13
Avg. tokens / file	324	76

TABLE III
TOKEN-LEVEL VECTOR GENERATION FINAL SETTINGS

Parameter	Value
Ancestors window size	2
Descendants window size	1
Siblings included	no
Output vector dimension	50

algorithm to generate token-level vectors for Java and Python. We then trained our model on the dataset we created and tuned its hyperparameters. Finally, we used the trained model to perform clone detection with files in our dataset which were not used for training in the previous step.

We run our experiments on a 12 cores Linux machine with 64GB of memory and an Nvidia Quadro P6000 GPU with 24GB of memory.

1) Token-level vectors generation: In order to train our model, we first need to generate token-level vectors using our tree-based skip-gram algorithm. For both Java and Python, we generated two different kinds of vocabularies:

1) Vocabulary without token values
2) Vocabulary of 10000 tokens with values

The vocabulary without token values only contains the type of each token, for example, `ImportStmt` in Java, or `FunctionDef` in Python, while the one with values also contains identifiers information.

We tried to learn the representation using a large set of values for the different hyperparameters we had. We tried window sizes from 0 to 5 for the ancestors, from 0 to 4 for the descendants, we tried to use siblings and we tried output dimensions of 10, 20, 50, 100 and 200. At this point, we only qualitatively evaluate the vector-representation by plotting a number of points on a 2D plane and looking if semantically similar nodes were close or not. If increasing the size of the representation did not present any significant benefit, we kept the smaller size. In Figure 5, we show a subset of some points plotted in 2D and clustered using k-means [24]. We can see that statements, expressions, and declarations are correctly clustered and semantics are somewhat preserved. For example, `ForStmt` and `WhileStmt` are exactly at the same point and literals are close in the vector space.

In Table III, we show the parameters we found to work best for generating token-level vector representations and which we actually used for the clone detection experiment.

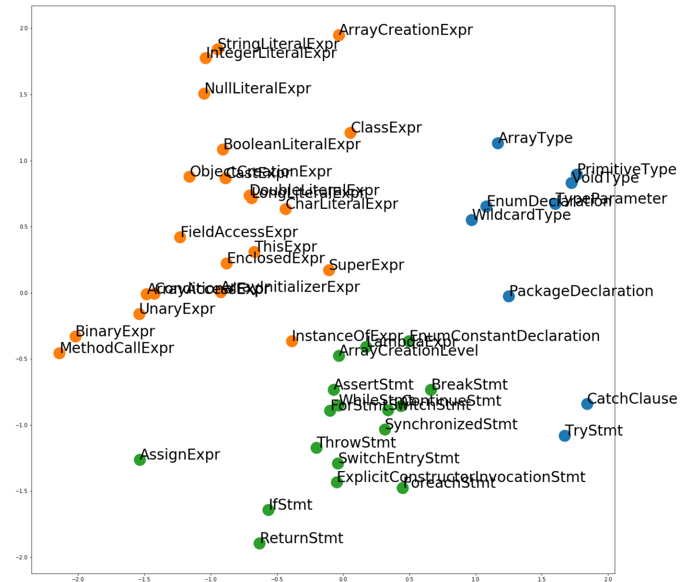

Fig. 5. Java token-level vectors projection in 2D

Increasing the window size too much seems to create too much noise, and did not yield better results. Likewise, we suspect that including the siblings generated more noise than signal when training our model.

2) Model training and testing: To evaluate our clone detection model, and see how our token-level representation affects its performance, we perform experiments on both the single-language clone detection task, where two input programs are written in Java, and the cross-language clone detection task where a program is written in Java and the other in Python.

It is worth noting that most clones being at best weakly type III, the dataset is difficult in nature and we did not manage to obtain a recall higher than 0.1 on Java clone detection using SourcererCC [4].

We prepared the dataset to train our model by splitting the dataset we described above into training set containing 80% of the data, the cross-validation set used to tune our hyperparameters containing 10% of the data, and finally, the test set used to give a final evaluation of our model. For both training, cross-validation, and test, we treat files implementing a solution to the same problem as clones and randomly choose n samples from files implementing a solution to a different problem to use as negative inputs to our model. We choose samples with a number of tokens close to the positive one, to make sure our model is not too biased by the input length. We make the number of negative samples vary during training but fix this number to 4 samples — giving us a dataset with 20% of clones — for cross-validation, in order to be able to compare the performance of the different models on the same input data. Below, we give more details about the different models we trained during the experiments.

Baseline. To show the importance of the AST structure when training the model, we create a baseline model which treats

TABLE IV
MODEL HYPERPARAMETERS

Name	Value
Token vector dimension	100
Encoder layer	bidirectional LSTM, stacked with 2 layers layer dimensions: 100 and 50
Classifier	single hidden layer, 64 units
Optimizer	RMSprop [25]
Epochs	50

source code as a sequential input for both token-level vector generation and clone detection. Concretely, the model used for clone detection in this baseline is the same as the one shown in Figure 4, but instead of using the AST transformer we simply feed the tokens in the source code sequentially, and the token-level vectors are learned by using the regular sequential skip-gram algorithm on tokens sequences.

Pre-trained token vectors. In this experiment, we use the model described in Section III and we initialized the weights of the embedding layer using the token-level vectors representation learned using our tree-based skip-gram algorithm.

Randomly initialized token vectors. In order to show the effect of the token-level vectors we learned using tree-based skip-gram, we use exactly the same model as the previous experiment but replace the learned representation by randomly initialized vectors and let our model learn the representation.

Pre-trained token vectors, no values. To see how the values of the tokens — the identifiers in the programs — influence the clone detection ability, we trained a model with a vocabulary containing only the token types, which means that the system cannot distinguish an identifier `x` from an identifier `y`. This reduces the size of the vocabulary to around 100.

In all our experiments, except the one where we exclude the values of the token, we used a vocabulary size of 10000 as increasing the size further did not significantly improve the results. This means that most identifiers which do not come up in the first 10000 tokens would not come up often enough in our dataset to be useful to our model. Other hyperparameters were also chosen experimentally, and we trained all the model described above with the same set of hyperparameters to ensure that the results were not influenced by other factors. We present the set of hyperparameters we used for training in Table IV and use these hyperparameters to measure the performance of our model on our test set. We show the results we obtained for cross-language clone classification in Table V and the results for Java clone classification in Table VI. Using our pre-trained vectors, we obtain an F1-score of 0.66 on the Java/Python clone classification task. This shows that our model is able to learn similarities between Java and Python programs, which responds positively to RQ1.

Our results show that for both cross-language and single-language clone detection, our model using pre-trained token vectors performs the best. Using the AST structure gives us around 12% F1-score and 15% precision improvement compared to our sequential model baseline. Using our token vectors pre-trained with our tree-based skip-gram algorithm gives us a 5% improvement on the F1-score for the cross-language task, and 3% improvement on the single-language task, which responds positively to RQ2. We assume that the improvement is greater for cross-language because it is simpler for the model to map tokens for code written in the same programming language, so there is less need for pre-training. Another important point about the benefit of our pre-trained vectors which is not reflected in these results is the time for which the model needs to be trained before converging. For example, in our Java/Python experiment, after only 10 epochs, our model using pre-trained vectors already has an F1-score of about 0.6 while the model using randomly initialized vectors have an F1-score of about 0.45. Finally, the results for the model not using token value is interesting because we get only an 8% decrease in the F1-score on the single-language clone detection task, while we get a 15% decrease on the cross-language detection task. The reason for this difference is likely that the model can more easily map the structure of the ASTs in a single-language context, making the need for the values of the tokens less important than in a cross-language context.

TABLE V
JAVA/PYTHON CLONE CLASSIFICATION RESULTS

Model	F1-score	Precision	Recall
Baseline	0.53	0.41	0.74
Pre-trained token vectors, no values	0.51	0.40	0.71
Randomly initialized token vectors	0.61	0.49	0.82
Pre-trained token vectors	0.66	0.55	0.83

TABLE VI
JAVA/JAVA CLONE CLASSIFICATION RESULTS

Model	F1-score	Precision	Recall
Baseline	0.65	0.50	0.92
Pre-trained token vectors, no values	0.69	0.56	0.90
Randomly initialized token vectors	0.74	0.65	0.85
Pre-trained token vectors	0.77	0.67	0.92

3) Clone detection experiment: In the previous experiment, we evaluated the similarity between the given code fragment and only five samples including one code clone. In this experiment, we evaluated the similarity among all the combinations of the given set of code fragments. We use our model to find clones in that set, which is how clone detection tools usually work. We use 500 randomly sampled files from our test set. As our system currently accepts only pairs of code fragments, it takes $\mathcal{O}(n^2)$ — where n is the number of input files — runs to perform clone detection we must run it on all the pairs of input files. To speed up the computation, we first precompute the vectors for all the input files, as described in III-D2 and then run the classifier part of our model on each pair of vectorized ASTs. We present the results we obtained in Table VII.

The recall results are as good as the one we obtained when testing our model, but the precision is an order of

TABLE VII
CROSS-LANGUAGE CLONE DETECTION RESULTS

Metric	Result
F1-score	0.32
Precision	0.19
Recall	0.90

magnitude lower than our previous results. As during the training phase we had only 4 negative samples per clone, we suspect that we did not manage to provide enough hard examples to train our model. To detect clone, we compare each code fragment to all the others, and the model therefore probably run in harder cases than the one it has seen during training time, thus increasing the number of false-positives. Further improvements to the negative sample selection process described in III-D1 should help improve the precision.

Overall, 90% of all the clones all correctly marked as positive but only 1 out of 5 of the clones marked as positive are actually clones. As a final response to RQ1, we conclude that although our prototype is able to learn similarities between programs written in Java and Python, it does not do it precisely enough for practical use yet.

C. Discussion

As briefly discussed in Section II, our system is mostly designed to detect cross-language clones where ASTs can be very fuzzily matched. Using our approach, we are able to detect clones across languages, such as the one presented in listings 1 and 2 — our system predicts these two programs are clones with 75% of confidence. On the other hand, as we are learning to match patterns in the structure of the programs, our system tends to mark programs with similar structures as clones, which negatively affects the precision score reported. For example, the programs in Listing 3 are an example of a false-positive we got when inspecting the results of our experiments. Although these two code fragments do not enter the type IV clone as defined in the literature, the codes do share many traits: reading a value from the standard input, initializing a variable to hold a temporary result, updating the result in a loop, and finally outputting a string conditionally depending on the value of the temporary result. Whether finding such patterns could really be helpful to help refactoring, or not, is an open question and would need a more thorough investigation to be answered.

More generally, the current literature about clone detection does not provide a clear taxonomy for cross-language clone detection. Types I to type III define clones by the similarities in the structure of the program. Even more granular classifications such as strongly and weakly type III [15] only really make sense in a single-language context for the same reason. This means that we have no way to classify cross-language clones as they all enter the type IV category of functional clones. There are at least a few points that we think are important to classify cross-language clone detection.

```
1  import java.util.*;
2
3  public class Main{
4    public static void main(String[] args){
5      Scanner sc = new Scanner(System.in);
6      int A = sc.nextInt(), B = sc.nextInt(), C =
           sc.nextInt();
7      boolean isFlag = false;
8      for(int i = 0; i < B ; i++){
9        if ( (A * i) % B == C){
10         isFlag = true;
11       }
12     }
13
14     if(isFlag){
15       System.out.println("YES");
16     } else{
17       System.out.println("NO");
18     }
19   }
20 }
```

```
1  N = int(input())
2  a = list(map(int, input().split()))
3
4  count2 = 0
5  count4 = 0
6
7  for ai in a:
8      if ai % 4 == 0:
9          count4 += 1
10     elif ai % 2 == 0:
11         count2 += 1
12
13 if count4*2 + count2 >= N or (count4*2+1) >= N:
14     print("Yes")
15 else:
16     print("No")
```

Listing 3. Clone pair false positive

First, do the code fragments implement the same algorithm? Two code fragments implementing the same functionality with different algorithms should be the weakest possible type of clone. Second, to what extent can the statements of the code fragments be matched. For example, in listings 1 and 2, the last `print` statements can be mapped directly, while the `for` loop statement is more ambiguous, although both loops do depend on some value read from the standard input. How to combine these properties needs further analysis, but is worth investigating, as we think improving this taxonomy will help to reason better about cross-language clones.

V. THREATS OF VALIDITY

The main threat of validity is that the data we used to test our model is vastly different from the data we could expect in a real-world system. As detailed in IV-A, we used data from competitive programming problems to train and test our model. Although the number of lines per code fragment in our dataset is relatively close to the one of a typical function, competitive programming has some particularities. For example, variable names are often less meaningful that in production code. Furthermore, the tasks solved by the program being extremely well-defined, it is easier for two

programs solving the same problem to look very similar than two functions in a typical code base. However, having different training and test datasets is more than common in machine learning and there are many solutions to this issue [26]. A possible solution could be to manually annotate a much smaller dataset and use it fine-tune the classifier of our model.

VI. Related works

In this section, we will discuss related work in two different categories: first, clone detection approaches for single-language and cross-language clone detection, then some other approaches to vector representation generation methods.

A. Clone detection approaches

Clone detection has been studied a lot in the literature, although most of the effort has been put into single-language clone detection. CCFinder [3] and more recently SourcererCC [4] present token-based techniques to detect code clones. These techniques work especially well for type III copy-paste induced code clones and are able to scale very well, as shown in [27]. Some other methods such as [28] also use somewhat similar approaches to detect plagiarism between programs. However, although the methods used are language agnostic in the sense they could be used for any programming language, they are not designed to work across programming languages, making their scope different from our work.

Deckard [13] presents a scalable AST based approach to clone detection, where a hash value is generated for subtrees in the AST and locality-sensitive hashing [29] is then applied to cluster code clones. The vector generation approach in this work is designed to work with programs written in the same language, and finding one which would work across programming languages is a research problem in itself.

In recent years, some clone detection work using deep learning techniques have emerged. In [30], the authors propose the RvNN model, which helps them improve the AST representation to achieve better performance for clone detection. In [31], the authors propose an alternative model which is built on tree-LSTMs [32] to represent ASTs for clone detection. Both works focus on Java clone detection and are mostly orthogonal to our work, as we could try to replace the encoder layer in our model by one of the proposed models.

Some approaches to cross-language clone detection have also been proposed but assume some sort of common intermediate representation between languages. In [33], the authors propose a system capable of detecting clones between C# and Visual Basic.NET by using CodeDOM[4] as an intermediate representation. The system is therefore not designed to perform clone detection across arbitrary languages such as Java and Python. Another approach, which is not directly designed for cross-language clone detection, is the one presented in [34], where clones are detected directly from the executable format. Although this approach would not work for our Java and Python example, it could potentially work across multiple programming languages if the same compiler backend (e.g. LLVM) were used to produce the binary.

[4]https://msdn.microsoft.com/library/system.codedom.aspx

B. Vector representation generation approaches

Many different approaches have been proposed in the literature to generate vector representation, either for words, tokens or nodes. The closest work to our tree-based skip-gram is the original skip-gram [16] algorithm, on which we based our method. As explained in Section III, while the skip-gram algorithm works sequentially on words in a sentence, our algorithm uses the structure of the tree to generate the context tokens of a particular target.

There also exist several approaches which are able to generate token-level representations for nodes in an arbitrary graph structure. node2vec [35] uses a custom of random walk mixing breadth-first search and depth-first search, while subgraph2vec [36] uses Weisfeiler-Lehman graph kernels [37] and an extension of the skip-gram algorithm to learn vector representations of rooted subgraphs. An important difference with our tree-based skip-gram is that our method focuses on learning vector representations in a tree topology. This allows us to have a clear distinction between ancestors and descendants, which is significant in the context of an AST. Some early work to learn token vector representations from ASTs can be found in [38], but this work only focuses on learning representations for the types of the nodes in the AST. As in our pre-trained token vectors with no values experiment, an identifier `x` and an identifier `y` are represented by the same token. Whether the same approach can be used when including identifiers is not clear.

VII. Conclusion

In this paper, we presented our cross-language clone detection method based on semi-supervised machine learning. For the unsupervised learning phase, we introduced the tree-based skip-gram algorithm to learn semantically meaningful representations of the program tokens. We also created a cross-language code clone dataset and used it to train and evaluate our model. We showed that our system is able to find interesting patterns across programs written in Java and Python.

Although our system is not yet designed to perform large scale clone detection, combining techniques such as deep hashing [39] and fast nearest neighbors search [40] should improve its speed enough to run at scale. The next step is to put this engineering effort into our system and to evaluate it empirically on real-world code bases to see to what extent it can be used to refactor large systems.

References

[1] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *SCHOOL OF COMPUTING TR 2007-541, QUEEN'S UNIVERSITY*, vol. 115, 2007.

[2] B. S. Baker, "A program for identifying duplicated code," *Computing Science and Statistics*, 1992.

[3] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002. [Online]. Available: http://dx.doi.org/10.1109/TSE.2002.1019480

[4] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 1157–1168. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884877

[5] P. Mayer, M. Kirsch, and M. A. Le, "On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers," *Journal of Software Engineering Research and Development*, vol. 5, no. 1, p. 1, Apr 2017. [Online]. Available: https://doi.org/10.1186/s40411-017-0035-z

[6] M. Flower, "Microservices — a definition of this new architectural term," https://martinfowler.com/articles/microservices.html, 2014, [Online; accessed 04-August-2018].

[7] B. Ed, M. Brian, and M. Mike, "How We Build Code at Netflix," https://medium.com/netflix-techblog/how-we-build-code-at-netflix-c5d9bd727f15, 2016, [Online; accessed 04-August-2018].

[8] E. Letuchy, "Facebook Chat," https://www.facebook.com/note.php?note_id=14218138919, 2008, [Online; accessed 04-August-2018].

[9] E. Reinhold, "Rewriting Uber Engineering: The Opportunities Microservices Provide," https://eng.uber.com/building-tincup/, 2016, [Online; accessed 04-August-2018].

[10] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, *Microservices: Yesterday, Today, and Tomorrow*. Cham: Springer International Publishing, 2017, pp. 195–216. [Online]. Available: https://doi.org/10.1007/978-3-319-67425-4_12

[11] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Refactoring support based on code clone analysis," in *Product Focused Software Process Improvement*, F. Bomarius and H. Iida, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 220–233.

[12] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[13] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 96–105. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.30

[14] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 368–. [Online]. Available: http://dl.acm.org/citation.cfm?id=850947.853341

[15] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sept 2014, pp. 476–480.

[16] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *CoRR*, vol. abs/1310.4546, 2013. [Online]. Available: http://arxiv.org/abs/1310.4546

[17] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a "siamese" time delay neural network," in *Proceedings of the 6th International Conference on Neural Information Processing Systems*, ser. NIPS'93. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, pp. 737–744. [Online]. Available: http://dl.acm.org/citation.cfm?id=2987189.2987282

[18] S. Chopra, R. Hadsell, and Y. LeCun, "Learning a similarity metric discriminatively, with application to face verification," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1. IEEE, 2005, pp. 539–546.

[19] J. Mueller and A. Thyagarajan, "Siamese recurrent architectures for learning sentence similarity." 2016.

[20] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[21] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*. IEEE, 2013, pp. 6645–6649.

[22] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Deepface: Closing the gap to human-level performance in face verification," in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, June 2014, pp. 1701–1708.

[23] W. M. Ulrich and P. Newcomb, *Information systems transformation: architecture-driven modernization case studies*. Morgan Kaufmann, 2010.

[24] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient k-means clustering algorithm: Analysis and implementation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, no. 7, pp. 881–892, Jul. 2002. [Online]. Available: http://dx.doi.org/10.1109/TPAMI.2002.1017616

[25] T. Tieleman and G. Hinton, "Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude," COURSERA: Neural Networks for Machine Learning, 2012.

[26] M. Sugiyama, N. D. Lawrence, A. Schwaighofer *et al.*, *Dataset shift in machine learning*, 2017.

[27] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "DéjàVu: a map of code duplicates on GitHub," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, oct 2017. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3152284.3133908

[28] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, and R. Robbes, "Language-independent clone detection applied to plagiarism detection," in *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on*. IEEE, 2010, pp. 77–86.

[29] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, ser. SCG '04. New York, NY, USA: ACM, 2004, pp. 253–262. [Online]. Available: http://doi.acm.org/10.1145/997817.997857

[30] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 87–98. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970326

[31] M. L. Huihui Wei, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 2017, pp. 3034–3040. [Online]. Available: https://doi.org/10.24963/ijcai.2017/423

[32] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *CoRR*, vol. abs/1503.00075, 2015. [Online]. Available: http://arxiv.org/abs/1503.00075

[33] N. A. Kraft, B. W. Bonds, and R. K. Smith, "Cross-language clone detection." in *SEKE*, 2008, pp. 54–59.

[34] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proceedings of the eighteenth international symposium on Software testing and analysis - ISSTA '09*. New York, New York, USA: ACM Press, 2009, p. 117. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1572272.1572287

[35] A. Grover and J. Leskovec, "Node2Vec: Scalable Feature Learning for Networks," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 855–864. [Online]. Available: http://doi.acm.org/10.1145/2939672.2939754

[36] A. Narayanan, M. Chandramohan, L. Chen, Y. Liu, and S. Saminathan, "subgraph2vec: Learning Distributed Representations of Rooted Sub-graphs from Large Graphs," jun 2016. [Online]. Available: http://arxiv.org/abs/1606.08928

[37] N. Shervashidze NINOSHERVASHIDZE, P. Schweitzer PASCAL, E. Jan van Leeuwen EJVANLEEUWEN, K. Mehlhorn MEHLHORN, and K. M. Borgwardt KARSTENBORGWARDT, "Weisfeiler-Lehman Graph Kernels," *Journal of Machine Learning Research*, vol. 12, pp. 2539–2561, 2011. [Online]. Available: http://www.jmlr.org/papers/volume12/shervashidze11a/shervashidze11a.pdf

[38] L. Mou, G. Li, Y. Liu, H. Peng, Z. Jin, Y. Xu, and L. Zhang, "Building Program Vector Representations for Deep Learning," sep 2014. [Online]. Available: http://arxiv.org/abs/1409.3358

[39] H. Zhu, M. Long, J. Wang, and Y. Cao, "Deep Hashing Network for Efficient Similarity Retrieval," in *Proceedings of the Thirtieth {AAAI} Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, {USA.}*, 2016, pp. 2415–2421. [Online]. Available: http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12039

[40] V. Hyvönen, T. Pitkänen, S. Tasoulis, E. Jääsaari, R. Tuomainen, L. Wang, J. Corander, and T. Roos, "Fast k-NN search," sep 2015. [Online]. Available: http://arxiv.org/abs/1509.06957