# Cross-language clone detection by learning over abstract syntax trees

Anonymous Author(s)

## ABSTRACT

While clone detection across programs written in the same programming language has been studied extensively in the literature, the task of detecting clones across multiple programming languages is not covered as well, and approaches based on comparison cannot be directly applied. In this paper, we present a clone detection method based on semi-supervised machine learning able to detect clone across programming languages. Our method uses an unsupervised learning approach to learn token-level vector representations and an LSTM-based neural network to predict if 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 more than 50000 code fragments written in Python and Java. We show that using our approach, we are able to detect functional clones missed by state-of-the-art clone detection tools.

## CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools*;

## KEYWORDS

Clone detection, Machine learning, Source code representation

**ACM Reference Format:**
Anonymous Author(s). 2018. Cross-language clone detection by learning over abstract syntax trees. In *Proceedings of The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Code clones are fragments of code, in a single or multiple programs, which are similar to each other. Code clones can emerge for various reasons, such as copy-pasting or implementing the same functionality as another developer. 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 copied. However, detecting these code clones is a difficult task and has been extensively researched in the literature. Some systems focus on finding clones inside a single project, while other task try to detect clones in larger ecosystems. Although there is a very large amount 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 well in the literature.

In real-world applications, systems are very often written using multiple programming languages and code duplication can therefore occur across programs written in different languages. In this paper, we present a semi-supervised machine learning based system capable of finding code clones across programming languages, and make the following contributions.

- We propose tree-based skipgram, an unsupervised machine learning algorithm used to learn semantically meaningful vector representations for tokens in a programming language and prove its efficiency in our experiments
- We present a cross-language clone detection system, provide an implementation supporting clone detection across Java and Python and show in our experiments that our tree-based skipgram algorithm improves its performance
- We create a cross-language code clones dataset containing around 50000 files written in Java and Python and annotations about which of the files are code clones

We make the source code for tree-based skipgram and our code clone detection system, as well as all the datasets we created for the experiments publicly available[1].

## 2 BACKGROUND

With the increase in the number of programming languages used in a single system, the ability to find code clones across languages has become more important. A typical example of a system using multiple programming languages is a system following a client-server architecture, where a server exposes its services to multiple clients by providing a remote API, usually accessible through a protocol such as HTTP or XML-RPC. Clients often implement similar functionalities to provide to the end users, but are typically written in different programming languages, as different platforms may not support the same programming languages — e.g. Android supports Java and Kotlin while iOS supports Swift and Objective-C. In listings 1 and 2, we show a simple motivating example which illustrates well the kind of issues that may occur in real-world applications. Here, we assume that a Python client (e.g. desktop application) and a Java client (e.g. Android application) both fetch a list of records from a remote API. Both clients need to display the list of records grouped by state, and therefore the grouping logic gets implemented by both clients. This logic could typically be extracted to the API side, removing the need to maintain the same functionality on all the clients. However, in practice, finding such refactoring opportunities is difficult as different clients are often developed by different teams. Providing insights on such potential

[1] Anonymized for blind review

```python
def group_records(records):
    result = {}
    for record in records:
        bucket = result.setdefault(record.state, [])
        bucket.append(record)
    return result
```

**Listing 1: Grouping function in Python**

```java
public Map<String, List<Record>> groupRecords(
    List<Record> records) {
  Map<String, List<Record>> grouped = new HashMap<>();
  for (Record record: records) {
    if (!grouped.containsKey(record.getState())) {
      grouped.put(record.getState(),
                  new ArrayList<Record>());
    }
    grouped.get(record.getState()).add(record);
  }
  return grouped;
}
```

**Listing 2: Grouping function in Java**

refactoring opportunities could help developers to avoid code duplication across the system, but it requires the ability to detect code clones across programming languages.

Although clone detection within a single programming language has been studied extensively, most methods are based on comparison between tokens or AST structures and are therefore not adapted for clone detection across programming languages. The only common tokens between the two code fragments above are the identifier `records`, the keyword **`for`** and the final **`return`**, and therefore, approaches relying on the number of tokens in common, such as SourcererCC [21] would not be able to find that these two methods are clones. The ASTs of these two code fragments are also very different in structure, and therefore AST-based approaches such as Deckard [7] using rule-based approach to cluster ASTs would not be able to find such a clone pair. Even for a simple example as the one presented above, there are an important number of rules that would need to be provided in order to match these two pieces of code. Given we already had some mapping between basic language constructs, such as method definitions and **`for`** statements, the `{}` literal in Python should be matched to the `HashMap` constructor, the `getState` getter should be matched to a property access, the condition to add a new entry to the map should be matched with `dict.setdefault` and `HashMap.add` should be matched with `list.append`. Creating and maintaining such rules for multiple languages would be impractical.

## 3 PROPOSAL

In this work, we propose a semi-supervised machine learning based system which is capable of detecting code clones across programming languages. A key component of this system is our token-level vectors generation algorithm, tree-based skipgram, 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

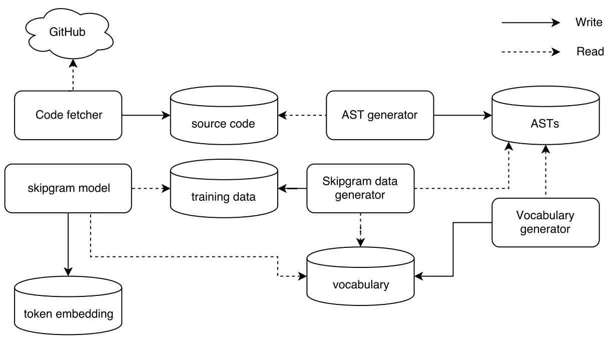

**Figure 1: Token-level vectors generation overview**

to map tokens — such as the `HashMap` constructor and the dictionary literal in the previous example — across languages.

In this section, we will first describe the token-levels vector generation process and then give an overview of the clone detection system as a whole.

### 3.1 Token-level vectors generation

The token-level vectors generation algorithm, tree-based skipgram, is based on the skipgram algorithm [14], but uses the structure of the AST to compute a vector representation of each token in a target programming language. While the skipgram algorithm treats its input as a sequence and generates the context of a particular target using the tokens around it, tree-based skipgram uses the tree structure to generate the context for a particular target. Tree-based skipgram is a base technique which can be used to help finding particular shapes of subtrees or compare subtrees in multiple ASTs.

The process for generating token-level vectors using tree-based skipgram 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 skipgram model with the generated target and context pairs

We show an overview of the token-level vectors generation process in figure 1. We will describe the algorithms to generate the vocabulary and generate the data to train the skipgram model, and give details about how we train the model. We will provide more details about the source code collection in Section 4.

#### 3.1.1 *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 have a type, 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

**Algorithm 1** Vocabulary generation algorithm

```
 1: function GenerateVocabulary(sourceFiles, includeValues, maxSize)
 2:     tokensCount ← empty map
 3:     for file in sourceFiles 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
```

higher than in NLP and we therefore want to avoid using a generic "unknown" token. Instead, we choose to use only the type of the token and to replace the token value by `null` when no token with the same type and value is found in the vocabulary, allowing us to at least keep some semantic information about the token. We therefore add the token type without its value to the vocabulary at generation time. We show how we generate the vocabulary $V$ in algorithm 1. In order to be able to fallback 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.

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

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 therefore 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 as shown in algorithm 2.

**Algorithm 2** Token lookup in vocabulary

```
1: function LookupTokenIndex(vocabulary, token)
2:     if (token.type, token.value) ∈ vocabulary then
3:         return vocabulary.indexOf((token.type, token.value))
4:     return vocabulary.indexOf((token.type, null))
```

**Algorithm 3** Data generation for skipgram 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:                 contextNodeIndex ← LookupTokenIndex(contextNode)
10:                 skipgramData +← (nodeIndex, contextNodeIndex)
11:     return skipgramData
```

*3.1.2 Skipgram data generation.* After generating the vocabulary, we generate data to train a skipgram 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 hyper-parameter. In our tree-based skipgram algorithm we take advantage of the topological information contained by the AST instead of working on a simple sequence of tokens. We therefore 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. We can therefore 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 a viable candidate 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, while a node will only have a single parent, it can have any number of children, and therefore, while having a window size of 3 for the ancestors would only generate 3 nodes in the context, if every descendant of a node had 5 children, a window size of 3 would generate $5^3 = 125$ nodes in the context, which would probably generate more noise than signal when trying to train the model. We therefore use two different parameters to control the window size of the ancestors and the window size of the descendant when generating the data to train our skipgram 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 3, 4 and 5, we describe the process we use to generate the data to train a skipgram model.

**Algorithm 4** 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 $\xleftarrow{+}$ 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 $\xleftarrow{+}$ sibling
12:     return contextNodes
```

**Algorithm 5** 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
```

Algorithm 3 takes as input a list of files written in the programming language for which we want to generate token embedding, the vocabulary extracted for this programming language and the parameters described above. It loops over all the nodes in the file, and uses algorithms 4 and 5 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 4 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 5 to find all the descendants of the node in the window given by the passed parameters, then find all the ancestors in the given window and finally add the siblings to the set of results if necessary. Algorithm 5 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.

*3.1.3 Training the skipgram model.* Once the data is generated using algorithms 3, 4 and 5, the last step needed to generate embedding is to actually train a skipgram model using the generated data. Algorithm 3 generates pairs of indexes which can be directly fed to a neural network, and there is therefore no need for further pre-processing. To train the model, the vocabulary used is the same as the one used to generate the skipgram data and the size of the embedding is a hyper parameter of the model. The model is trained using a negative sampling objective $NEG$ shown in equation 2 as given in [14].

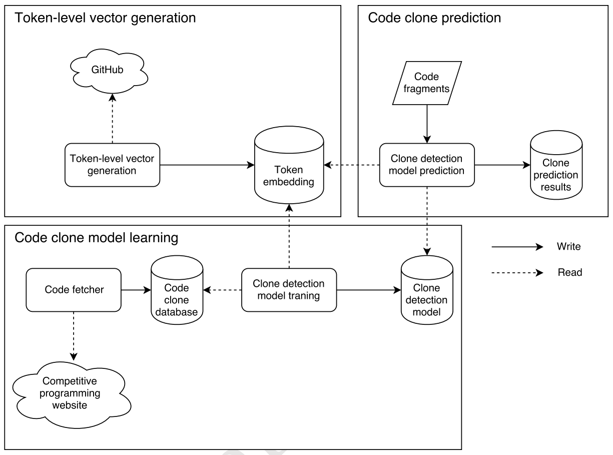
Token-level vector generation
GitHub
Token-level vector generation
Token embedding
Code clone prediction
Code fragments
Clone detection model prediction
Clone prediction results
Code clone model learning
Code fetcher
Code clone database
Clone detection model traning
Clone detection model
Competitive programming website
Write
Read

**Figure 2: Clone detection system overview**

$$NEG = \log \sigma\left({v'_{w_O}}^T \cdot v_{w_I}\right) + \sum_{i=1}^{k} \log \sigma\left(-{v'_{w_i}}^T \cdot v_{w_I}\right) \quad (2)$$

In equation 2, $v$ are the weights of the hidden layer, which will be used as embedding, $v'$ are the weights of the output layer, $w_I$ is an input token and $w_O$ is a token in its context. $w_i$ are "noise" tokens which are randomly sampled from the vocabulary using a noise distribution $P_n$, for which we use a unigram distribution raised to the power of 3/4, for reasons described in [14]. The dimension of $v$ is $\mathbb{R}^{|V| \times d_w}$ where $|V|$ is the size of the vocabulary and $d_w$ is the size of the embedding, which is a hyper-parameter of the model.

## 3.2 Clone detection system

Our system uses a semi-supervised learning approach, where the vector representation of the tokens are computed in an unsupervised fashion using the tree-based skipgram described above and the learned representations is used when feeding the ASTs tokens to the supervised model used to perform the clone detection. We show an overview of the system in figure 2.

It is important to note that our system is designed to detect semantic clones, also called functional clones or type IV clones, which are two code fragments that implement the same functionality, but are not syntactically or structurally similar [3, 10, 15]. This differs from type III clones, which make assumptions on the structure of the code fragments [1, 11]. We will discuss this further at the end of Section 4.

In the rest of this section, we will describe the model we use to perform the clone detection. We will give details about the dataset we use to train the model in Section 4.

*3.2.1 Model overview.* Our clone detection model uses an end-to-end learning approach, where two programs are fed to a model which predicts the probability of these being clones. The model is composed of the following layers.

(1) AST transformer layer — transforms an AST into a sequence of index

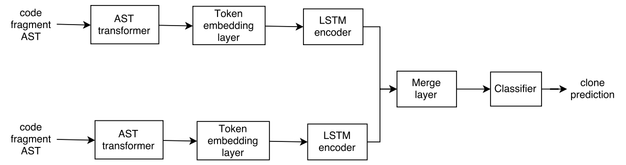

**Figure 3: Model overview**

(2) Token embedding layer — maps each index into a vector in $\mathbb{R}^d$
(3) Encoder layer — transforms a sequence of vectors into a single vector
(4) Merge layer — merges vectors from two ASTs in a single vector
(5) Classifier — predicts if the encoded vector represents a pair of clone or not

We show how these layers are connected together in figure 3.

*3.2.2 AST transformer layer.* The AST transformer layer takes an AST as input and maps it to a vector of integers where each value is an index of the input AST programming language vocabulary. More formally, if we let $\mathcal{T_L}$ be the set of all possible ASTs for source code written in programming language $\mathcal{L}$ and $V_\mathcal{L}$ its vocabulary, the AST transformer $f_a^\mathcal{L}$ will be a function with the following definition where $m$ depends on the size of the input tree and the algorithm used for $f_a^\mathcal{L}$.

$$f_a^\mathcal{L} : \mathcal{T_L} \rightarrow \mathbb{N}^m \tag{3}$$

In our implementation of $f_a^\mathcal{L}$, we traverse the tree using a depth-first, pre-order tree traversal. At each node in the tree, we look up the index of the node token in the vocabulary using algorithm 2 and we append the index to the output vector. In this case, the output is all the AST nodes linearized by the traversal, and $m$ will therefore equal the number of nodes in the input AST.

*3.2.3 Token embedding layer.* When we transform an AST in to a vector using the above approach, we obtain a vector of indexes. The embedding layer of the model transforms a vector in $\mathbb{N}^m$ into a matrix in $\mathbb{R}^{m \times d_w}$ where $d_w$ is the dimension of the embedding — a hyper parameter of the model. The embedding layer for a programming language $\mathcal{L}$ is therefore a function $f_w^\mathcal{L}$ with the following dimensions.

$$f_w^\mathcal{L} : \mathbb{N}^m \rightarrow \mathbb{R}^{m \times d_w} \tag{4}$$

Such embedding are actually represented by a matrix $W^\mathcal{L}$ which has dimensions of $\mathbb{R}^{|V| \times d_w}$ where $|V|$ is the size of the vocabulary, and $d_w$ is the dimension of the embedding. The matrix can be randomly initialized and learned from scratch by the model or initialized using the vector representations learned using the tree-based skipgram algorithm presented in Subsection 3.1. We experimentally show how using the representation learned by tree-based skipgram influences the performance of the model in Section 4.

*3.2.4 Encoder layer.* As discussed previously, the output of the embedding layer will be a matrix of dimension $\mathbb{R}^{m \times d_w}$. As we want encoded ASTs to have the same dimensions, the only requirement of this layer is that the output dimension is independent of the input dimension, so that with $d_e$ being a hyper-parameter of the model the encoder $f_e^\mathcal{L}$ is defined as follow.

$$f_e^\mathcal{L} : \mathbb{R}^{m \times d_w} \rightarrow \mathbb{R}^{d_e} \tag{5}$$

In our implementation, we use a stacked bidirectional LSTM model. We choose the dimension of $d_e$ experimentally by trying values between 20 and 200. We later show the exact values we used in table 4.

*3.2.5 Merge layer.* Up to now, we have focused on how to encode a single AST into a vector in $\mathbb{R}^{d_e}$. The role of the merge layer is to combine the vectors of the two input ASTs in a single vector. It is defined by a function $f_m$ with the following type, where $n$ depends on the function used.

$$f_m : \mathbb{R}^{d_e} \times \mathbb{R}^{d_e} \rightarrow \mathbb{R}^n \tag{6}$$

In our model, the merge layer uses the combination of the additive and multiplicative distances of the two vectors, as described in [26]. Given $h_L$ the output of the encoder for the first code fragment and $h_R$ the output of the encoder for the second code fragment, the output $h$ of the merge layer is defined by the following equations, where $\odot$ is the element-wise multiplication operator.

$$\begin{aligned} h_\times &= h_L \odot h_R \\ h_+ &= |h_L - h_R| \\ h &= W^{(\times)} h_\times + W^{(+)} h_+ \end{aligned} \tag{7}$$

With this implementation, $W^{(\times)}$ and $W^{(+)}$ are parameters of the model and are both matrices with dimension $\mathbb{R}^{d_o \times d_e}$ where $d_o$ is a hyper-parameter of the model. The dimensions of the different results will therefore be $h_\times \in \mathbb{R}^{d_e}$, $h_+ \in \mathbb{R}^{d_e}$ and $h \in \mathbb{R}^{d_o}$, hence $n = d_o$.

*3.2.6 Classifier.* Once we obtain a single vector containing information about the two input code fragments, we use a classifier to predict if the inputs were code clones or not. The classifier is a function $f_p$ defined as follow, where the output can be interpreted as a probability.

$$f_p : \mathbb{R}^n \rightarrow [0, 1] \tag{8}$$

In our model, we use a feed-forward neural network and will later show the parameters we used for it in table 4.

*3.2.7 Loss function.* To train our model, we use the binary cross entropy loss function, defined as follow.

$$L(y, \hat{y}) = y \log \hat{y} + (1 - y) \log(1 - \hat{y}) \tag{9}$$

In equation 9, $y$ is the expected output — 1 if the two input code fragments are clones and 0 otherwise — and $\hat{y}$ is the output of our model. We show the hyper-parameters used for training in table 4.

## 4 EXPERIMENTS

In our experiments, we focus on the following research questions.

RQ 1 can we use the structure of the AST to learn a semantically meaningful representation of the different tokens in a programming language
RQ 2 can we use this representation to improve the ability to detect code clones across programming languages

**Table 1: 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 |

**Table 2: Clone detection dataset metrics**

| | Java | Python |
|---|---|---|
| Avg. files / problem | 36 | 41 |
| Files count | 20,828 | 23,792 |
| Lines count | 958,246 | 312,353 |
| Tokens count | 6,744,391 | 1,810,085 |

We decompose our experiments in two main steps: we first train a model implementing the tree-based skipgram algorithm to learn token-level vector representation for Java and Python. We then train our clone detection model using the token-level representation we obtained in the first step.

In order to conduct our experiments, we created two different datasets, a large corpus of source code used to generate token-level vectors and an annotated dataset of clone pairs written in Java and Python to train our clone detection model. We will give further details about these two dataset below.

## 4.1 Datasets creation

*4.1.1 Token-level vectors generation dataset.* Tree-based skipgram 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]. For Python, as we could not find any organization with a sufficient amount of source code, we chose projects which fulfilled the following conditions.

- Size between 100KB and 100MB
- Non-viral license (e.g. MIT, BSD)
- Not forks

We ordered the results by 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, the produced AST may vary slightly and we therefore decided to use only Python 3 for this experiment. We show some metrics of our dataset in table 1.

*4.1.2 Code clones dataset.* To train our clone detection model, the dataset creation is more difficult as the model is supervised and we therefore need information about two programs in the corpus are clones or not. The dataset needs fulfills the following properties.

(1) Multiple code fragments should implement the same functionality
(2) Information on whether two code fragments implement the same functionality must be included
(3) Dataset should contain code fragments written in at least 2 programming languages

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

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

We found that competitive programming websites fulfill all the above properties. The solution to a single problem is implemented by a large number of persons in many different languages. 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 at least type IV code clones. The easier the problem is, the higher the probability of code fragments implementing the solution to the same problem have to be very similar to each other, and to therefore be closer to type III clones. 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 2.

To create the dataset, we used code from a famous competitive programming website[3]. As our implementation currently only supports Java and Python, we fetched data for these two programming languages. We restricted the data to only programs that were accepted by the website judging system — meaning that the programs actually implemented the solution to the given problem — in order to have the type IV code clone guarantee. We collected code for a total of 576 different problems and give some metrics about the dataset in table 2.

## 4.2 Experiments and Results

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

*4.2.1 Token-level vectors generation experiment.* We first present the details of the experiment we run with our tree-based skipgram algorithm to generate Java and Python token-level vectors.

The first step when generating token-level vectors is to generate a vocabulary for the given programming language. We will here present the results for the different vocabularies we generated. 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. In figure 4, we show the relation between the rank of a token in the vocabulary, and its number of appearances. We use a log scale for both axes. An interesting property we find is that, this

[3]Anonymized for blind review

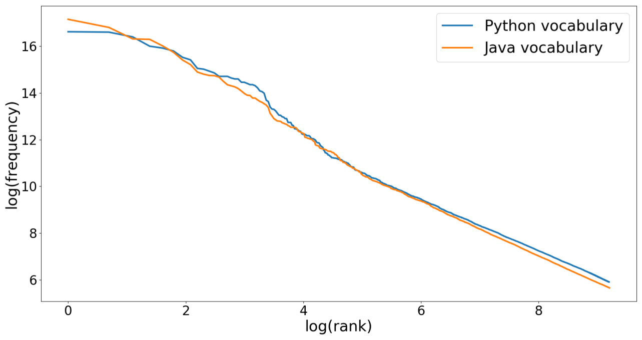

**Figure 4: Relation between tokens rank and frequency**

relation and by extension the distribution of the frequency tokens in a programming language roughly follows the Zipf's law [18]. Given the distribution of the token frequency in both programming languages follow closely the frequency found in natural languages, we can expect that a model close to skipgram, which works well with natural languages could also work here.

Once we create our vocabulary, we want to assign a vector in $\mathbb{R}^{d_w}$ to each token in our vocabulary. Given the purpose of generating these representations is to encode a program so that we can detect clone, the main requirement of our representation is that tokens which are semantically similar, such that for example `while` and `for` have a small distance in the resulting vector space, while unrelated tokens have a large distance.

Given the unsupervised nature of the model, it is difficult to evaluate quantitatively the quality of the results. While some methods exist to evaluate word vector representation in natural languages context [22], the technique uses the nature of the language and is therefore not directly applicable to programming languages. We are not aware of such a method to evaluate vector representations in programming languages context. At this stage, we therefore evaluated qualitatively the results by manually inspecting the relation between different tokens, and judging of the quality of the representation using these relationship. In particular, when learning token-level vectors for vocabulary where token do not have values, we try to cluster the tokens using k-means[9] and look at how much the clusters make sense — for example, are statements and expressions correctly clusters.

We tried to learn the representation using a large set of values for the different parameters 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. When increasing the size of the representation did not present any significant benefit, we kept the smaller size. In table 3, we show the parameters we found to work best for generating token-level vector representations.

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. In figure 5, we show the results we obtained when projecting the learned vectors in a 2 dimensions space using PCA[24].

**Table 3: Token-level vector generation final settings**

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

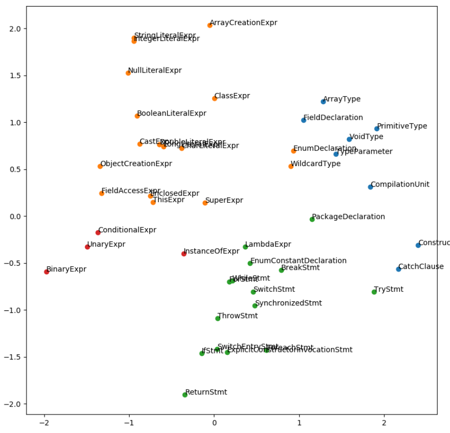

**Figure 5: Java token vectors 2D projection for vocabulary without values**

We only plotted a subset of the points in the vocabulary to keep the figure readable.

Although, given the nature of the experiment, the evaluation of the result is fragile, this does partially affirm RQ 1.

*4.2.2 Clone detection experiments.* 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.

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 hyper parameters 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,

Table 4: Model hyper-parameters

| 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 [27] |
| Epochs | 50 |

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 perform experiments using a baseline model which treats 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 3, but instead of using the AST transformer we simply feed the tokens in the source code sequentially, and the embeddings are learned by using the regular skipgram algorithm on tokens sequences.

**Pre-trained token vectors** In this experiment, we use the model described in Section 3 and we initialized the weights of the embedding layer using the token-level vectors representation learned using our tree-based skipgram algorithm.

**Randomly initialized token vectors** In order to show the effect of the token-level vectors we learned using tree-based skipgram, we use exactly the same model as the previous experiment but replace the learned representation by randomly initialized vectors and let the end-to-end 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, meaning that most identifiers which do not come up in the first 10000 tokens do not come up often enough in our dataset to be useful to our model. Other hyper-parameters were also chosen experimentally, and we trained all the model described above with the same set of hyper-parameters to ensure that the results were not influenced by other factors. We present the set of hyper-parameters we used for training in table 4, the results we obtained for cross-language clone detection in table 5 and the results for Java clone detection in table 6.

Our results show that for both cross-language and single-language clone detection, our model using pre-trained token vectors performs best. Using the AST structure gives us around 12% F1-score and 15% precision improvement compared to our sequential model baseline.

Table 5: Java/Python clone detection 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 6: Java/Java clone detection 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 |

Using our token vectors pre-trained with our tree-based skipgram algorithm gives us a 5% improvement on the F1-score for cross-language task, and 3% improvement on the single language task, which responds to RQ 2. 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, reducing the 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, as we get only a 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.

*4.2.3 Comparison to SourcererCC.* In this experiment, we want to see if our system is able to detect clones which are not detected by current clone detection approaches. We choose to compare our method to SourcerCC[21], as it is the tool published the most recently with its source code publicly available. In the above experiment, we sampled 4 non-clones code pairs for each clone pair when evaluating our model. However, clone detection tools normally do not work by taking code pairs but search for clones in a set of code fragments. Our system currently only works by taking code fragments pairs as input, and we therefore need to run $O(n^2)$ comparisons to detect clones in the same way as the tool we are comparing our results to.

To run our experiment, we used the files from our test dataset, and run our tools using all the pairs combinations, as well as SourcererCC on the sampled files to detect clones within these files. To run the experiment, we select the model which performed the best in the above experiment according to the results presented

**Table 7: Clone detection results comparison**

| | Threshold | F1-score | Precision | Recall |
|---|---|---|---|---|
| Our method | 0.6 | 0.21 | 0.12 | 0.85 |
| SourcererCC | 0.7 | 0.05 | 0.63 | 0.03 |

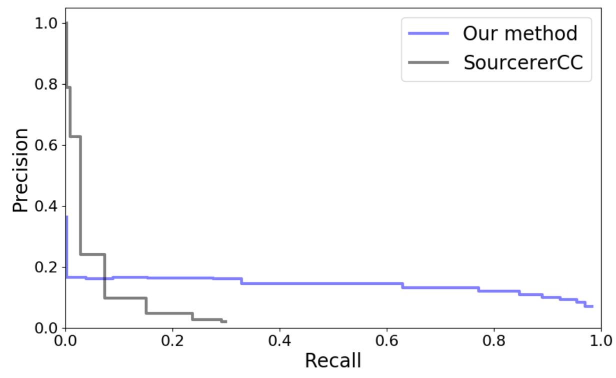

**Figure 6: Clone detection precision-recall curve**

in table 6. We generate all the possible code pairs of the files in our test set, and use our model to predict whether each of these pairs were clones or not. Using our predictions, which are probability-like values between 0 and 1, we set multiple thresholds to whether the clone pair is a code clone are not. If the threshold is closer to 0, we accept pairs that are less likely to be clones and therefore trade precision for recall, on the other hand if the threshold is closer to 0.9, we are stricter for which pairs to accept and therefore trade recall for precision. We try a range of values for this threshold and compute the F1-score, precision and recall for each of them.

After running the experiment with our model, we use the same dataset to run the experiment with SourcererCC. SourcererCC uses a threshold to decide whether it considers code fragments to be clones or not. If the value of the threshold is 0.5, it means that at least 50% of the tokens must be shared between the code fragments to be considered as clones. The higher the threshold, the more the code fragments need to be similar to be considered as clones, therefore when the threshold becomes higher, the precision increases and the recalls drops.

In table 7, we show the results we obtained for both systems, and plot the precision-recall curve obtained by adjusting the sensitivity of the systems in figure 6. Our method and SourcererCC have very different characteristics and are therefore difficult to actually compare — our method has a very high recall but a poor precision, while SourcererCC has a good precision but a low recall. We will here try to give some explanation about the results we obtained. Our dataset is mostly composed of types IV clones, with probably a small amount of clones which would enter the type III clone category. SourcererCC is a token-based clone detection tool and is mostly design to detect type III clones, and given the nature of its algorithm should be particularly efficient for clones generated from copy-pasting. We believe that the poor results of SourcererCC on this dataset comes from its nature — clones in the dataset being introduced by different programmers implementing the same functionality, which generate very different types of clones. We think that the poor precision results with our approach comes from two different factors. The main important cause for this issue is that our tool is currently only able to predict if two code fragment are clones are not, and to be able to run this experiment, we therefore need to predict $O(n^2)$ samples. The amount of clones in these samples is therefore less than 1%, which vastly lower than the 20% of clones we were using when training the model. This change in the training and test distributions therefore tends to increase the number of false positives. This issue is left to future work and we give more details in Section 6.

## 4.3 Discussion

As briefly discussed in Section 3, our system is mostly designed to detect functional clones. This is an important property, as directly comparing the structure of programs written in different programming languages is very difficult. 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 clone with 75% of confidence. However, the notion of functional clone is broad, and two code fragments implementing the same functionality with completely different algorithms are also classified as functional clones [19]. Our system does try to match patterns in the input code fragments and is therefore not adapted for such cases. On the other hand, as we are learning to match patterns in the structure of the programs, our system tends to mark programs with very similar structures as clones, which negatively impacts the precision score reported. For example, the programs in listing 3 is an example of a false-positive we got when inspecting our experiments results. 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 the result, updating the result in a loop, and finally outputting the result. Whether finding such patterns could really be helpful, for example to help refactoring, or not is an open question and would need 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, and even more granular classification such as strongly and weakly type III [25] only 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. 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. Another potential property is up to what point can the statements of the code fragments can be matched. For example, in listings 1 and 2, the initialization and for statement have a clear one-to-one mapping, while the mapping between the body of the for statements is more ambiguous. 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.

```java
public static void main(String[] args) {
  Scanner sc = new Scanner(System.in);
  int n = sc.nextInt();
  int ans = Integer.MAX_VALUE;
  for (int i = 1; i <= n; i++) {
    int j = n / i;
    ans = Math.min(ans, Math.abs(i - j) + n - i * j);

  }
  System.out.println(ans);
}
```

```java
public static void main(String[] args){
  Scanner read = new Scanner(System.in);
  int x = Integer.parseInt(read.nextLine());
  int c = 0;
  int sum = 0;
  while (sum < x){
    c++;
    sum += c;
  }
  System.out.println(c);
}
```

**Listing 3: Clone pair false positive**

## 5 RELATED WORK

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

### 5.1 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 skipgram is the original skipgram [14] algorithm, on which we based our method. As explained in Section 3, while the skipgram 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 [4] uses a custom of random walk mixing breadth first search and depth first search, while subgraph2vec [17] uses Weisfeiler-Lehman graph kernels [23] and an extension of the skipgram algorithm to learn vector representations of rooted subgraphs. An important difference with our tree-based skipgram is that our method focuses on learning vector representations in a tree topology, allowing 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 [16], but this work only focuses on learning representations for the types of the nodes in the AST, so as for 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.

### 5.2 Clone detection approaches

Clone detection has been studied a lot in the literature, although most of the effort has been put in single-language clone detection. CCFinder [8] and more recently SourcererCC [21] 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 [13]. 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 [7] presents a scalable AST based approach to clone detection, where a hash value is generated for subtrees in the AST and locality-sensitive hashing [2] 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 [28], the authors propose the RvNN model, which helps them improve the AST representation to achieve better performance for clone detection. In [5], the authors propose an alternative model which built on tree-LSTMs [26] to represent ASTs for clone detection. Both work 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 model to improve our results.

Some approaches to cross-language clone detection have also been proposed, but generally assume some sort of common intermediate representation between languages. In [12], 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 [20], 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.

## 6 CONCLUSION

In this paper, we presented a cross-language clone detection system based on semi-supervised machine learning. For the unsupervised learning phase, we introduced the tree-based skipgram 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 obtained promising results for clone detection across programming languages and showed that our system is able to find interesting patterns across programs.

Although our system is not yet designed to perform large scale clone detection, combining techniques such as deep hashing [29] and fast nearest neighbors search [6] should improve its speed enough to run at scale. The next step is therefore to put this engineering effort and evaluate empirically our system on real-world systems to see up to what point the results can be used for refactoring in large systems.

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

## REFERENCES

[1] S Bellon, R Koschke, G Antoniol, J Krinke, and E Merlo. 2007. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering* 33, 9 (sep 2007), 577–591. https://doi.org/10.1109/TSE.2007.70725

[2] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive Hashing Scheme Based on P-stable Distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry (SCG '04)*. ACM, New York, NY, USA, 253–262. https://doi.org/10.1145/997817.997857

[3] Neil Davey, Neil Davey, Paul Barson, Simon Field, and Ray J Frank. 1995. The Development of a Software Clone Detector. *International Journal of Applied Software Technology* 1, 3 (1995), 219– 236. https://doi.org/10.1.1.41.7548

[4] Aditya Grover and Jure Leskovec. 2016. Node2Vec: Scalable Feature Learning for Networks. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. ACM, New York, NY, USA, 855–864. https://doi.org/10.1145/2939672.2939754

[5] Ming Li Huihui Wei. 2017. 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*. 3034–3040. https://doi.org/10.24963/ijcai.2017/423

[6] Ville Hyvönen, Teemu Pitkänen, Sotiris Tasoulis, Elias Jääsaari, Risto Tuomainen, Liang Wang, Jukka Corander, and Teemu Roos. 2015. Fast k-NN search. (sep 2015). arXiv:1509.06957 http://arxiv.org/abs/1509.06957

[7] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 96–105. https://doi.org/10.1109/ICSE.2007.30

[8] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Softw. Eng.* 28, 7 (July 2002), 654–670. https://doi.org/10.1109/TSE.2002.1019480

[9] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. 2002. An Efficient k-Means Clustering Algorithm: Analysis and Implementation. *IEEE Trans. Pattern Anal. Mach. Intell.* 24, 7 (July 2002), 881–892. https://doi.org/10.1109/TPAMI.2002.1017616

[10] R. Komondoor and S. Horwitz. 2003. Effective, automatic procedure extraction. In *Proceedings - IEEE Workshop on Program Comprehension*, Vol. 2003-May. IEEE Comput. Soc, 33–42. https://doi.org/10.1109/WPC.2003.1199187

[11] Rainer Koschke, Raimar Falke, and Pierre Frenzel. 2006. Clone Detection Using Abstract Syntax Suffix Trees. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*. IEEE Computer Society, Washington, DC, USA, 253–262. https://doi.org/10.1109/WCRE.2006.18

[12] Nicholas A Kraft, Brandon W Bonds, and Randy K Smith. 2008. Cross-language Clone Detection.. In *SEKE*. 54–59.

[13] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: a map of code duplicates on GitHub. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (oct 2017), 1–28. https://doi.org/10.1145/3133908

[14] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. *CoRR* abs/1310.4546 (2013). arXiv:1310.4546 http://arxiv.org/abs/1310.4546

[15] Gilad Mishne and Maarten de Rijke. 2004. Source Code Retrieval Using Conceptual Similarity. In *Coupling Approaches, Coupling Media and Coupling Languages for Information Retrieval (RIAO '04)*. LE CENTRE DE HAUTES ETUDES INTERNATIONALES D'INFORMATIQUE DOCUMENTAIRE, Paris, France, France, 539–554. http://dl.acm.org/citation.cfm?id=2816272.2816322

[16] Lili Mou, Ge Li, Yuxuan Liu, Hao Peng, Zhi Jin, Yan Xu, and Lu Zhang. 2014. Building Program Vector Representations for Deep Learning. (sep 2014). arXiv:1409.3358 http://arxiv.org/abs/1409.3358

[17] Annamalai Narayanan, Mahinthan Chandramohan, Lihui Chen, Yang Liu, and Santhoshkumar Saminathan. 2016. subgraph2vec: Learning Distributed Representations of Rooted Sub-graphs from Large Graphs. (jun 2016). arXiv:1606.08928 http://arxiv.org/abs/1606.08928

[18] Steven T. Piantadosi. 2014. ZipfâĂŹs word frequency law in natural language: A critical review and future directions. *Psychonomic Bulletin & Review* 21 (2014), 1112–1130. Issue 5. https://doi.org/10.3758/s13423-014-0585-6

[19] Chanchal Kumar Roy and James R. Cordy. 2007. A Survey on Software Clone Detection Research. *SCHOOL OF COMPUTING TR 2007-541, QUEENâĂŹS UNIVERSITY* 115 (2007).

[20] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. 2009. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis - ISSTA '09*. ACM Press, New York, New York, USA, 117. https://doi.org/10.1145/1572272.1572287

[21] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 1157–1168. https://doi.org/10.1145/2884781.2884877

[22] Tobias Schnabel, Igor Labutov, David Mimno, and Thorsten Joachims. 2015. Evaluation methods for unsupervised word embeddings. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 298–307.

[23] Nino Shervashidze NINOSHERVASHIDZE, Pascal Schweitzer PASCAL, Erik Jan van Leeuwen EJVANLEEUWEN, Kurt Mehlhorn MEHLHORN, and Karsten M Borgwardt KARSTENBORGWARDT. 2011. Weisfeiler-Lehman Graph Kernels. *Journal of Machine Learning Research* 12 (2011), 2539–2561. http://www.jmlr.org/papers/volume12/shervashidze11a/shervashidze11a.pdf

[24] Jonathon Shlens. 2014. A Tutorial on Principal Component Analysis. *CoRR* abs/1404.1100 (2014). arXiv:1404.1100 http://arxiv.org/abs/1404.1100

[25] J Svajlenko, J F Islam, I Keivanloo, C K Roy, and M M Mia. 2014. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 476–480. https://doi.org/10.1109/ICSME.2014.77

[26] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. *CoRR* abs/1503.00075 (2015). arXiv:1503.00075 http://arxiv.org/abs/1503.00075

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

[28] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep Learning Code Fragments for Code Clone Detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 87–98. https://doi.org/10.1145/2970276.2970326

[29] Han Zhu, Mingsheng Long, Jianmin Wang, and Yue Cao. 2016. Deep Hashing Network for Efficient Similarity Retrieval. In *Proceedings of the Thirtieth {AAAI} Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, {USA.}*. 2415–2421. http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12039