Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code *

Hui-Hui Wei and Ming Li
National Key Laboratory for Novel Software Technology, Nanjing University
Collaborative Innovation Center of Novel Software Technology and Industrialization
Nanjing 210023, China
{weihh, lim}@lamda.nju.edu.cn

Abstract

Software clone detection, aiming at identifying out code fragments with similar functionalities, has played an important role in software maintenance and evolution. Many clone detection approaches have been proposed. However, most of them represent source codes with hand-crafted features using lexical or syntactical information, or unsupervised deep features, which makes it difficult to detect the functional clone pairs, i.e., pieces of codes with similar functionality but differing in both syntactical and lexical level. In this paper, we address the software functional clone detection problem by learning supervised deep features. We formulate the clone detection as a supervised learning to hash problem and propose an end-to-end deep feature learning framework called CDLH for functional clone detection. Such framework learns hash codes by exploiting the lexical and syntactical information for fast computation of functional similarity between code fragments. Experiments on software clone detection benchmarks indicate that the CDLH approach is effective and outperforms the state-of-the-art approaches in software functional clone detection.

1 Introduction

Software clones are introduced when developers reuse code by the copy-paste-modify operations [Roy and Cordy, 2007], or when a developer implements a functionality that is very similar to an existing one [White *et al.*, 2016]. Since software clone may easily lead to the injection of software defects or infringement of copyright [Baker, 1995; Brixtel *et al.*, 2010], software clone detection, aiming to identify similar code fragments, has attracted significant attention.

Software clone can be categorized into four different types based on different levels of similarity [Roy and Cordy, 2007]: **Type-1**: identical code fragments in addition to variations in comments and layout; **Type-2**: apart from Type-1 clones, identical code fragments except for different identifier names and literal values; **Type-3**: apart from Type-1 and -2 clones, syntactically similar code that differ at the statement level. The code fragments have statements added, modified and/or removed with respect to each other; **Type-4**: syntactically dissimilar code fragments that implement the same functionality. Among these four types of clones, effectively detecting Type-4 clones (also known as *functional* clones) is most challenging since the implementation of the same functionality may be quite different (e.g., summation implemented with for-loop and recursion) and it would be difficult to measure the *functional* similarity simply based on the appearance of the code fragments.

Many methods have been proposed to detect software clones. The key idea is to hand-craft certain similarity between two code fragments by exploiting either lexical information or syntactical information of the codes. NICAD [Roy and Cordy, 2008] applies slight transformations to code and measures similarity by comparing sequences of text. CCFinderX [Kamiya *et al.*, 2002] and SourcererCC [Sajnani *et al.*, 2016] treat source codes as bags of tokens and compare subsequences to detect clones. Deckard [Jiang *et al.*, 2007] introduces AST (Abstract Syntax Tree) to measure the structure similarity of two code fragments. NICAD, SourcererCC and CCFinderX are typical lexicon-based approaches which only consider the similarity in lexical level of code fragments and ignore the syntactical information. Thus, these methods may be able to successfully detect the lexical clones (e.g., Type-1 and Type-2) and would be ineffective for detecting Type-3 and Type-4 clones in most cases. On the other hand, Deckard is a typical syntax-based approach which uses structure information of software while the lexicon information of the code has not been taken into consideration. Although the syntactical information may be helpful in detecting Type-3 clones, it may still not be effective in detecting Type-4 clones. All these methods exhibit their incapability in detecting the Type-4 functional clones, while Type-4 clones are much more than the other types of clones (e.g., more than 98% in *BigCloneBench* dataset as shown in Table 3) in practice. In order to detect most of the clones, detecting the Type-4 clones should be explicitly considered.

Intuitively, the functional similarity could be measured if the functional behaviors of the code fragments can be carefully modeled. To achieve this goal, latent features that characterizing the functionality of the code could be learned by simultaneously considering both the lexical information and

*This research was supported by NSFC (61422304, 61272217).

syntactical structures of code fragments. Recently, autoencoder [Socher *et al.*, 2011] has been applied to learn latent features for source codes [White *et al.*, 2016]. Since autoencoder works in an unsupervised way, although this method may leverage the lexical and syntactical information to learn semantical latent features, the goal of feature learning is to reach an informative and compact representation of code fragments rather than identify the functional similarity between code fragments. With the explicit guidance for functional similarity, the similar code fragments based on such representations may not similar in their functions. Therefore, to learn latent representations in the purpose of capturing the similarities in terms of functional behaviors of code fragments, the feature learning should be conducted in a *supervised* manner, where the similarity in functional behaviors is used as supervision to bias the feature learning.

In this paper, we address software functional clone detection, i.e., Type-4 clone detection, by learning supervised deep features. The basic idea is to exploit deep learning methods to extract deep features automatically, meanwhile use the similarity in functional behaviors as the supervised information to guide the deep feature learning process. Besides, we use learning to hash [Kong and Li, 2012] to further transform the real-valued representations to binary hash codes in the purpose of improving the detection efficiency and saving storage space. Specifically, we formulate the clone detection problem as a supervised deep feature learning problem via a pairwise labels where clone pairs are regarded as positive examples and non-clone pairs are negative pairs. We propose an end-to-end deep feature learning framework for clone detection, namely CDLH (Clone Detection with Learning to Hash), which simultaneously learn hash functions and representations of code fragments via AST-based LSTM (Long Short-Term Memory) to take into account both the lexical and syntactical aspects of source codes. Experimental results on software clone detection benchmarks indicate that CDLD is effective and outperforms the state-of-the-art approach in detecting the functional clones of code fragments.

The contribution of this paper lies in three folds:

- We formulate clone detection as a supervised deep learning problem by introducing pairwise labels, which can effectively solve Type-4 clone detection, as well as the other types of clone. To the best of our knowledge, no previous methods adopt similar formulation of the problem and can achieve similar performance as our method.
- We propose CDLH, an end-to-end deep feature learning framework which simultaneously learns representations of code fragments and parameters of hash functions. The learned binary representation enables a fast computation of the similarity comparison of the code fragments and a low cost of storage space .

The rest of the paper is organized as follows: Section 2 presents the problem definition. Section 3 introduces the proposed approach CDLH. Section 4 reports the experimental results. Finally, Section 5 concludes the paper.

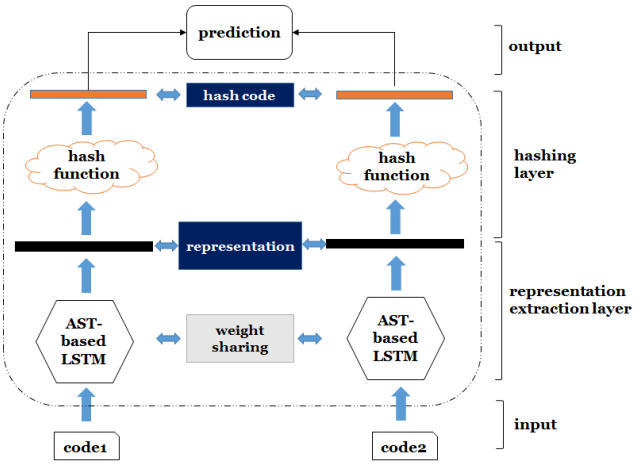


Figure 1: The overall architecture of CDLH.

2 Problem Definition

Given n code fragments $\{\mathrm{C}_1, \cdots, \mathrm{C}_n\}$ where C_i is the i-th raw code fragment, and pairwise labels to indicate whether two code fragments belong to a clone pair or not: $y_{i,j} = 1$ if $(\mathrm{C}_i, \mathrm{C}_j)$ is a clone pair, $y_{i,j} = -1$ if $(\mathrm{C}_i, \mathrm{C}_j)$ is not a clone pair, and $y_{i,j} = 0$ if their relation is undefined, then the training set is represented by a set of triplets $\mathcal{D} = \{(\mathrm{C}_i, \mathrm{C}_j, y_{i,j}) | i, j \in [n], i < j\}$, where $[n] = \{1, 2, \cdots, n\}$, and our goal is to learn a function Φ which maps any pairs of raw code fragments to $\{-1, 1\}$ to decide whether they belong to a clone pair.

Specifically, we simultaneously learn the non-linear representation mapping function ϕ for the code fragment representation (i.e., $\boldsymbol{z}_i = \phi(\mathrm{C}_i), \forall i \in [n]$ transforming raw code fragments $\{\mathrm{C}_i\}^n$ to representations $\{\boldsymbol{z}_i\}^n$) and a hash function $\psi : \mathbb{R}^d \rightarrow \{-1, 1\}^m$ mapping the d dimensional representation into the Hamming space (i.e. $\psi(\boldsymbol{z}_i) = [h_1(\boldsymbol{z}_i), h_2(\boldsymbol{z}_i), \ldots, h_m(\boldsymbol{z}_i)], \forall i \in [n]$ encoding $\{\boldsymbol{z}_i\}^n$ into binary hash codes $\{\boldsymbol{a}_i\}^n$), so that the Hamming distance between the hash codes of two clone pairs can be as small as possible, and the distance between the hash codes of none-clone pairs can be as large as possible. Given a pair of hash codes $(\boldsymbol{a}_i, \boldsymbol{a}_j)$, we apply a common function $g(\boldsymbol{a}_i, \boldsymbol{a}_j) = \mathbb{I}\left(\sum_{k=1}^{m} 1/4 * (a_{i,k} - a_{j,k})^2 \leq thr\right)$ to decide whether they belong to a clone pair [1], where $\mathbb{I}(.)$ is the indicator function which returns 1 if the condition is satisfied and returns -1 otherwise. Then we have $\Phi(\mathrm{C}_i, \mathrm{C}_j) = g(\psi(\phi(\mathrm{C}_i)), \psi(\phi(\mathrm{C}_j)))$.

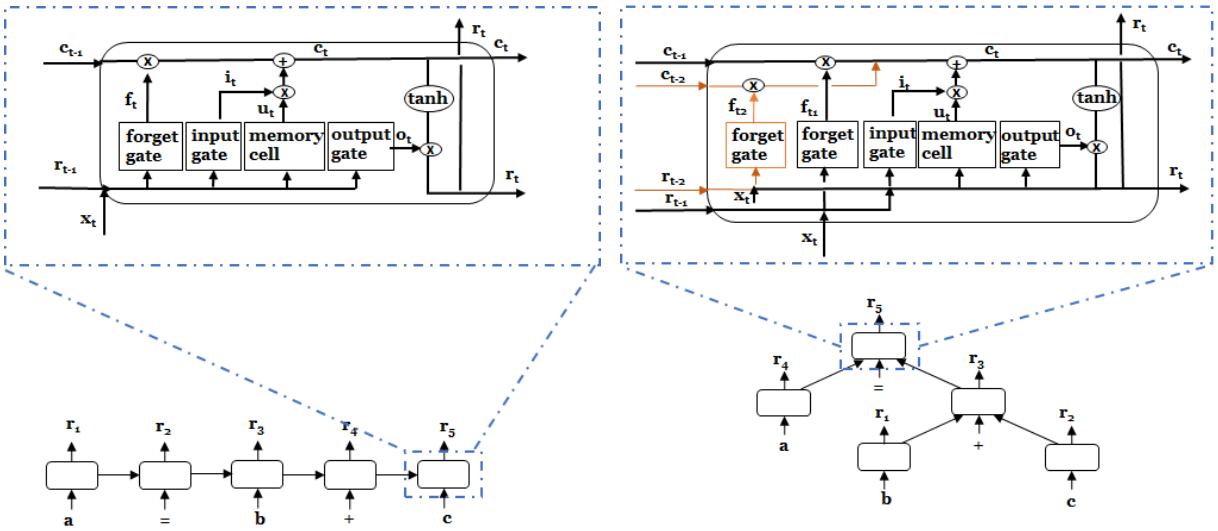
3 The Proposed Approach: CDLH

In this section, we present our proposed approach CDLH, which is an end-to-end learning approach that unifies the representation extraction layer and hashing layer into an integration. Figure 1 summarizes the overall architecture of CDLH.

3.1 The General Framework

As shown in Figure 1, CDLH mainly contains two parts: representation extraction layer and hashing layer. The inputs of CDLH are raw code fragments. CDLH first transforms them

[1] Usually, the threshold thr is set as 2.



(a) Traditional LSTM for statement `a=b+c;`

(b) AST-based LSTM for statement `a=b+c;`

Figure 2: Comparison between traditional LSTM and AST-based LSTM.

into ASTs, then uses AST-based LSTM to obtain the real-valued representation for each code fragment (i.e., representation extraction layer). After that, CDLH tries to learn hash functions to encode those representations into binary hash codes (i.e., hashing layer), so that code fragments belonging to a clone pair can be close to each other in terms of hamming distance, otherwise they should be far away. These two parts are integrated into one architecture, learning to map raw code fragments to pairwise labels.

Specifically, we try to simultaneously learn the representation mapping ϕ and the hash function ψ. In order to force the hash codes for clone pairs to be close to each other, and those for none-clone pairs to be far away, similar to [Lin *et al.*, 2014] we define the optimization problem as follows:

$$\min_{W,\phi} \sum_{i=1}^{n}\sum_{j=1}^{n} |y_{i,j}| \left[y_{i,j} - \frac{1}{m}\sum_{k=1}^{m} h_k\left(\phi(\mathrm{C}_i)\right) h_k\left(\phi(\mathrm{C}_j)\right) \right]^2, \quad (1)$$

where $h_k\left(\phi(\mathrm{C}_i)\right) = \mathrm{sign}(\boldsymbol{w}_k^\top \phi(\mathrm{C}_i) + b_k)$, b_k is a bias term, $W = \{\boldsymbol{w}_1, \cdots, \boldsymbol{w}_m, b_1, \cdots, b_m\}$.

In order to learn the representation mapping function ϕ which incorporates both the lexical and syntactical information of source codes, we design AST-based LSTM, which will be introduced in the next subsection. For parameter learning, we use AdaDelta [Zeiler, 2012] for optimization.

3.2 Representation Extraction with AST-based LSTM

In this section, we present AST-based LSTM to learn the representation mapping function ϕ, which can incorporate the lexical and syntactical information of source codes, since it leverages the AST to capture structure information of code fragments and LSTM to extract the semantic information carried by lexical tokens of source codes. An illustration of AST-based LSTM is shown in Figure 2(b).

AST extracts the syntactic information (or structure) of a code fragment, which will much facilitate software clone detection. For example, given two simple statements `int_1 = int_2 + int_3;` and `float_1=float_2-float_3; // a simple subtraction operation`, they are quite different as two lexical sequences. However, they share the same AST structure. Actually, these two expressions is a Type-2 clone pair, and through comparing their ASTs this conclusion can be draw correctly. Besides, from the definition of Type-3 clone, we can expect that using AST is also very helpful, since syntactically similar codes can be detected by comparing their ASTs even with different statements.

Before introducing AST-based LSTM, we first briefly introduce the traditional LSTM [Zaremba and Sutskever, 2014]. The lower part of Figure 2(a) shows how traditional LSTM processes expression `a = b + c;`. It treats the expression as a chain structure and extracts the representation in lexical level. The upper part of Figure 2(a) shows the detailed structure of a single traditional LSTM unit. It can be observed that each unit contains a forget gate f_t which is the weight of remembering old information, an input gate i_t which is the weight of acquiring new information, a memory cell c_t which records the cell state vector, an output gate o_t which is the weight of outputing memory cell, and a hidden state r_t which is the representation for tokens seen till now.

Then we apply LSTM units on AST structure. The lower part of Figure 2(b) presents AST-based LSTM for the same expression as that in Figure 2(a). Unlike traditional LSTM which processes the expression in a sequential way, AST-based LSTM processes the expression following the AST

structure. For example, in Figure 2(b) AST-based LSTM firstly processes subexpression 'b+c' and obtains $\boldsymbol{r}_3$, then uses $\boldsymbol{r}_3$ and $\boldsymbol{r}_4$ to obtain $\boldsymbol{r}_5$, which is also the representation for the expression 'a = b + c;'. The upper part of Figure 2(b) shows the detailed structure of an AST-based LSTM unit, in which the differences from the traditional LSTM unit are emphasized with orange color, i.e., the addition of cell state ($\boldsymbol{c}_{t-2}$) and hidden state ($\boldsymbol{r}_{t-2}$) of another previous unit, together with the forget gate ($\boldsymbol{f}_{t2}$) for it. Similar to traditional LSTM unit, each AST-based LSTM unit also contains an input gate, a memory cell and an output gate, whereas different from standard LSTM unit which only has one forget gate for its previous unit, an AST-based LSTM unit contains multiple forget gates (i.e., each for one of its children). Similar to [Tai *et al.*, 2015], an AST-based LSTM unit is updated as following:

$$\begin{aligned}
\boldsymbol{i} &= \sigma(W_i\boldsymbol{x} + \sum_{l=1}^{L} U_{il}\boldsymbol{r}_l + \boldsymbol{b}_i),\\
\boldsymbol{f}_l &= \sigma(W_f\boldsymbol{x} + U_{fl}\boldsymbol{r}_l + \boldsymbol{b}_f),\ l = 1, 2, \ldots, L\\
\boldsymbol{o} &= \sigma(W_o\boldsymbol{x} + \sum_{l=1}^{L} U_{ol}\boldsymbol{r}_l + \boldsymbol{b}_o),\\
\boldsymbol{u} &= \tanh(W_u\boldsymbol{x} + \sum_{l=1}^{L} U_{ul}\boldsymbol{r}_l + \boldsymbol{b}_u),\\
\boldsymbol{c} &= \boldsymbol{i} \odot \boldsymbol{u} + \sum_{l=1}^{L} \boldsymbol{f}_l \odot \boldsymbol{c}_l,\\
\boldsymbol{r} &= \boldsymbol{o} \odot \tanh(\boldsymbol{c}),
\end{aligned} \tag{2}$$

where $\boldsymbol{x}$ is the input word embedding of the corresponding token, L is the number of children, $\boldsymbol{f}_l$ $(l = 1, 2, \ldots, L)$ are L forget gates for children of the AST node, l is index number for its children, $W_i, W_f, W_o, W_u, U_{il}, U_{fl}, U_{ol}, U_{ul}$ are weight matrices, $\boldsymbol{b}_i, \boldsymbol{b}_f, \boldsymbol{b}_o, \boldsymbol{b}_u$ are bias vectors, σ is the logistic sigmoid function and $\odot$ is element-wise multiplication.

Notice that the number of children L varies for different nodes of different ASTs, which may cause problem in parameter-sharing. To facilitate subsequent procedure, we transform ASTs to binary trees whose nodes only contain 2 or 0 children. The process contains two steps: 1) split nodes with more than 2 children, and generate a new right child together with the old left child as its children, and then put all children except the leftmost as the children of this new node. Repeat this operation in a top-down way until only nodes with 0, 1, 2 children left; 2) combine nodes with 1 child with its child. Now only nodes with 0 or 2 children remain and the AST could be transformed into a binary tree.

For implementation, we compute the hidden states and cell states for each AST-based LSTM unit recursively in a bottom-up way. Then the output from the AST root (e.g., $\boldsymbol{r}_5$ in Figure 2(b)) is the representation for the code fragment.

4 Experiment

In this section we conduct experiments on real-world datasets and report the results. Specifically, we first compare CDLH with state-of-the-art clone detection approaches, then we validate the effectiveness of representations extracted by CDLH using AST-based LSTM. All our experiments for CDLH are complemented on a NVIDIA K80 GPU server.

Table 1: Overall information for datasets.

Datasets	Language	# code fragments	% clone pair	AVG length
BigCloneBench	JAVA	9,134	13.97	28.60
OJClone	C	7,500	0.07	35.25

4.1 Experimental Setting

We conduct our experiments on two real-world datasets covering different programming languages: a widely used benchmark dataset for clone detection *BigCloneBench* [Svajlenko *et al.*, 2014] (with JAVA code fragments), and *OJClone* from a pedagogical programming open judge (OJ) system[2] (with C code fragments).

Specifically, BigCloneBench consists of projects from 25,000 systems, covers 10 functionalities including 6,000,000 true clone pairs and 260,000 false clone pairs. Note that all those clone types are given by domain experts. We discard code fragments without any tagged true and false clone pairs, and use the remaining 9,134 code fragments.

OJClone contains 104 programming problems together with different source codes students submit for each problem [Mou *et al.*, 2016]. In OJClone, two different source codes solving the same programming problem are considered as a clone pair, since they realize the same functionality, and at least belong to Type-3 clone. In the experiment, we select the first 15 programming problems, and for each problem, there are 500 source code files. Note that two source code fragments for the same problem belong to a clone pair, and those for different problems are none clone pairs. For OJClone, we do not have experts to distinguish different clone types.

For BigCloneBench, a code fragment is a method, and for OJClone, a code fragment is a file. In order to extract AST structure, we use javalang[3], a pure python library for working with Java source code, to parse JAVA codes to ASTs, and apply pycparser[4] to parse C files to ASTs. To obtain word embeddings for tokens of original code fragments, we use word2vec[5] to generate word embeddings of length 100 for both datasets. For the length of hash code, we perform experiments with different length: $\{8, 16, \cdots, 48\}$ [6]. The overall information for datasets is listed in Table 1.

4.2 Comparison with Clone Detection Approaches

To evaluate the performance of CDLH, we compare CDLH with several state-of-the-art clone detection approaches as follows:

[2]http://programming.grids.cn

[3]https://github.com/c2nes/javalang

[4]https://pypi.python.org/pypi/pycparser/

[5]http://radimrehurek.com/gensim/models/word2vec.html

[6]Due to the limited space, the experimental results exhibited in the following sections are with 32 bit hash code. For other length, similar results can be concluded.

Table 2: Precision, recall and F1 comparison of all clone detection approaches.

Approaches	BigCloneBench			OJClone		
	P	R	F1	P	R	F1
Deckard	0.93	0.02	0.03	0.99	0.05	0.10
DLC	0.95	0.01	0.01	0.71	0.00	0.00
SourcererCC	0.88	0.02	0.03	0.07	0.74	0.14
CDLH	0.92	0.74	0.82	0.47	0.73	0.57

- Deckard [Jiang *et al.*, 2007], a popular syntactical-based clone detection tool.
- Approach proposed by [White *et al.*, 2016], which is the latest approach extracting unsupervised deep features using autoencoder. Later we will name it DLC for short.
- SourcererCC [Sajnani *et al.*, 2016], a state-of-the-art lexical-based clone detector.

We compare the CDLH with these clone detection approaches in terms of: precision (P), recall (R), F1 value, and F1 with respect to various clone types. Since apart from the overall detection performance measured by precision, recall and F1 value, we also wish to know the performance across various clone types, especially for Type-4 clone.

Table 2 shows the values of overall precision, recall and F1. We can see that CDLH significantly outperforms other clone detection methods in terms of F1 value. In general, it achieves much higher recall than other baselines, although precision may be sacrificed a little. This observation is mainly due to that CDLH is able to deal with Type-4 clone, and other baselines almost fail to detect any Type-4 clone, whereas Type-4 clone takes up more than *98%* over all clone types according to Table 3. Such result indicate that it is beneficial to leverage the supervised information to learn suitable (latent) feature vectors of code fragments for the clone detection tasks. Other clone detection approaches get poor F1 values on BigCloneBench, due to their poor F1 values with respect to Type-4 clone which are nearly 0 according to Table 4. Among which DLC is an approach using deep learning techniques, meaning that using latent features extracted by deep learning techniques also fails to detect Type-4 clone without the guidance of supervised information.

Without detailed clone types, it can be inferred that most clone pairs of OJClone belong to Type-3 or Type-4 clone, since two submitted files for OJ systems are hardly identical or only differ in identifier names, variable values, etc. The compared clone detection methods also achieve poor performance on OJClone. For Deckard and DLC, the reason is similar to that on dataset BigCloneBench. SourcererCC obtains high recall on OJClone, while relatively low precision, since it treats many code fragment pairs as clone pairs, which leads to many false positives. This happens because OJClone consists of code written by students, the name of variables and comments are less normalized, even though two code fragments contains many overlap tokens such as `a`, `b`, `i`, `j`, it is also unsuitable to treat them as clones.

As only BigCloneBench is tagged with various clone types, we measure F1 value with respect to various clone types only on BigCloneBench. In [Svajlenko *et al.*, 2014], due to the ambiguity in Type-3 and Type-4 clones, the author divide clone types for BigCloneBench into 5 categories: Type-1, Type-2, Strong Type-3 with similarity range in [0.7, 1), Mid Type-3 in [0.5, 0.7), and Type-4 in [0.5, 0). The percentage of various clone types in BigCloneBench is exhibited in Table 3, we can see that Type-4 clone pairs take up *98%* over all clone types, Mid Type-3 take up *1%* and left less than *1%* for other clone types. It shows the importance of Type-4 clone again: *most of clones are Type-4 clone.* Table 4 shows the F1 values with respect to these 5 clone types. It can be observed that CDLH can find over 80% Type-4 clones, whereas the F1 values for other baselines on Type-4 clone detection are nearly 0, which strongly supports our previous statement. For other types, CDLH also performs well, which validates the effectiveness of our approach.

Table 3: Number percentage of various clone types for BigCloneBench.

Type-1	Type-2	Strong Type-3	Mid Type-3	Type-4
0.005	0.001	0.002	0.010	0.982

Table 4: Comparison of F1 values with respect to various clone types on BigCloneBench, the best performed across each clone type is emphasized with boldface.

Clone types	Deckard	DLC	SourcererCC	CDLH
Type-1	0.73	**1.00**	0.94	**1.00**
Type-2	0.71	0.97	0.93	**1.00**
Strong Type-3	0.54	0.60	0.77	**0.94**
Mid Type-3	0.21	0.03	0.10	**0.88**
Type-4	0.02	0.00	0.00	**0.81**

4.3 The Effectiveness of Representation Extraction

In previous section we validate the superiority of CDLH on Type-4 clone detection by using supervised information. In this section we further validate the influence of representation extraction layer of CDLH, i.e., the effectiveness of AST-based LSTM by comparing other representation extraction approaches for programming language. For this purpose we use DLC, a state-of-the-art clone detection approach which extracts unsupervised deep features, P-CNN derived from subnetwork of NP-CNN [Huo *et al.*, 2016], a state-of-the-art deep method which handles programming languages, and a well-known sentence embedding tool Doc2Vec[7], together with CDLH to extract representations for code fragments.

To validate the effectiveness of representations extracted by CDLH, we fix the hashing layer of CDLH and compare following approaches with CDLH:

- representations extracted by DLC, and hashing layer of CDLH, we name it DLC+H for short;
- representations extracted by P-CNN, and hashing layer of CDLH, we name it P-CNN+H in the following;
- representations extracted by Doc2Vec, and hashing layer of CDLH, we name it Doc2Vec+H for short;

[7] http://radimrehurek.com/gensim/models/doc2vec.html

Table 5: Precision, recall and F1 comparison of all.

Approaches	BigCloneBench			OJClone		
	P	R	F1	P	R	F1
CDLH	0.92	0.74	0.82	0.47	0.73	0.57
DLC+H	0.24	0.60	0.35	0.07	0.01	0.02
P-CNN+H	0.20	0.87	0.33	0.07	0.04	0.05
Doc2Vec+H	0.25	0.96	0.39	0.07	0.03	0.05
CDLH+LSH	0.64	0.45	0.53	0.25	0.72	0.37
DLC+LSH	0.14	0.68	0.24	0.07	0.33	0.11
P-CNN+LSH	0.13	0.00	0.01	0.07	0.02	0.03
Doc2Vec+LSH	0.18	0.17	0.17	0.07	0.36	0.11

Table 6: Comparison of F1 values with respect to various clone types for approaches using learning to hash techniques on BigCloneBench, the best performed across each clone type is emphasized with boldface.

Clone types	CDLH	DLC +H	P-CNN +H	Doc2Vec +H
Type-1	**1.00**	**1.00**	0.95	**1.00**
Type-2	**1.00**	0.39	0.33	0.40
Strong Type-3	**0.94**	0.32	0.82	0.40
Mid Type-3	**0.88**	0.32	0.75	0.40
Type-4	**0.81**	0.34	0.33	0.39

To eliminate the effect of learning to hash has on representations extracted by CDLH, and to show that without learning to hash CDLH is still a good way to extract binary hash codes for programming language, we use unsupervised data-independent hashing LSH instead of learning to hash method used by CDLH, and compare following methods:

- representations extracted by CDLH, and LSH, we name it CDLH+LSH;
- representations extracted by DLC, and LSH, we name it DLC+LSH;
- representations extracted by P-CNN, and LSH, we name it P-CNN+LSH;
- representations extracted by Doc2Vec, and LSH, we name it Doc2Vec+LSH;

We treat above approaches as two groups according to hashing techniques they used.

Table 5 shows the precision, recall and F1 comparison of above approaches. We can see that CDLH+H and CDLH+LSH achieve highest F1 values in each group, which validates that representations extracted by AST-based LSTM can obtain good performance despite of the hashing techniques used. Approaches using representations extracted by DLC, P-CNN and Doc2Vec can either obtain low precision or low recall no matter using learning to hash technique or not, suggesting that representations extracted by them are less distinguished compared with representations extracted by AST-based LSTM. Approaches using LSH rather than learning to hash can obtain relatively inferior performance. Since LSH is an unsupervised data-independent hashing method, it can not learn task-specific pattern from training data and lack of guidance of supervised information. Even in this case, the representations extracted by CDLH still outperform other representations, which confirms the effectiveness of representations extracted by CDLH.

Table 7: Comparison of F1 values with respect to various clone types for approaches using LSH on BigCloneBench, the best performed across each clone type is emphasized with boldface.

Clone types	CDLH +LSH	DLC +LSH	P-CNN +LSH	Doc2Vec +LSH
Type-1	**1.00**	**1.00**	0.00	0.73
Type-2	**1.00**	0.25	0.03	0.28
Strong Type-3	**0.77**	0.01	0.01	0.27
Mid Type-3	**0.69**	0.23	0.01	0.25
Type-4	**0.52**	0.23	0.01	0.16

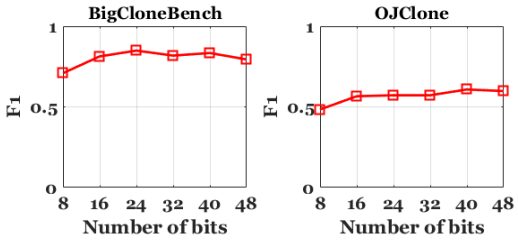


Figure 3: The influence of hash code length on CDLH.

F1 values with respect to various clone types for approaches using learning to hash techniques and LSH are listed in Table 6 and Table 7 respectively. As observed, CDLH and CDLH+LSH outperform other approaches in each group, they obtain high F1 values for Type-4 clone, together with other clone types. This further validates the effectiveness of the representations extraction layer of CDLH.

4.4 Sensitivity to Hyper-Parameter

In previous experimental settings, we use code length 32 for learned binary hash codes. Now we study the influence of different length of hash code on clone detection performance of CDLH, measured by F1 value. Figure 3 shows the F1 values of CDLH with respect to different code lengths, which ranges from 8 to 48. We can find that the overall performance of CDLH is not sensitive to hash code length in this range.

5 Conclusion

In this paper, we address the software functional clone detection problem by learning supervised deep features. We formulate the clone detection as a supervised learning to hash problem and propose an end-to-end deep feature learning framework called CDLH for functional clone detection, where an AST-based LSTM is used to exploit the lexical and syntactical information and the supervision on the functional similarity is used to guide the feature learning. Experiments on software clone detection benchmarks indicate that the CDLH approach is effective and outperforms the state-of-the-art approaches in software functional clone detection.

References

[Baker, 1995] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the 2nd Working Conference on Reverse Engineering*, pages 86–95, Toronto, Canada, 1995.

[Baxter *et al.*, 1998] Ira D. Baxter, Andrew Yahin, Leonardo Mendonça de Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the 1998 International Conference on Software Maintenance*, pages 368–377, Bethesda, Maryland, USA, 1998.

[Brixtel *et al.*, 2010] Romain Brixtel, Mathieu Fontaine, Boris Lesner, Cyril Bazin, and Romain Robbes. Language-independent clone detection applied to plagiarism detection. In *Proceedings of the 10th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 77–86, Timisoara, Romania, 2010.

[Huo *et al.*, 2016] Xuan Huo, Ming Li, and Zhi-Hua Zhou. Learning unified features from natural and programming languages for locating buggy source code. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, pages 1606–1612, New York, NY, USA, 2016.

[Jiang *et al.*, 2007] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. DECKARD: scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, pages 96–105, Minneapolis, MN, USA, 2007.

[Kamiya *et al.*, 2002] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[Kong and Li, 2012] Weihao Kong and Wu-Jun Li. Isotropic hashing. In *Proceedings of the 26th Annual Conference on Neural Information Processing Systems*, pages 1655–1663, Lake Tahoe, Nevada, USA, 2012.

[Lin *et al.*, 2014] Guosheng Lin, Chunhua Shen, Qinfeng Shi, Anton van den Hengel, and David Suter. Fast supervised hashing with decision trees for high-dimensional data. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1971–1978, Columbus, OH, USA, 2014.

[Mou *et al.*, 2016] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, pages 1287–1293, Phoenix, Arizona, USA, 2016.

[Roy and Cordy, 2007] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queens School of Computing TR*, 541(115):64–68, 2007.

[Roy and Cordy, 2008] Chanchal Kumar Roy and James R. Cordy. NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, pages 172–181, Amsterdam, The Netherlands, 2008.

[Sajnani *et al.*, 2016] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1157–1168, Austin, TX, USA, 2016.

[Socher *et al.*, 2011] Richard Socher, Jeffrey Pennington, Eric H. Huang, Andrew Y. Ng, and Christopher D. Manning. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 151–161, Edinburgh, UK, 2011.

[Svajlenko *et al.*, 2014] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal Kumar Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution*, pages 476–480, Victoria, BC, Canada, 2014.

[Tai *et al.*, 2015] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing*, pages 1556–1566, Beijing, China, 2015.

[White *et al.*, 2016] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 87–98, Singapore, 2016.

[Zaremba and Sutskever, 2014] Wojciech Zaremba and Ilya Sutskever. Learning to execute. *CoRR*, abs/1410.4615, 2014.

[Zeiler, 2012] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.