Positive and Unlabeled Learning for Detecting Software Functional Clones with Adversarial Training

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 is an important problem for software maintenance and evolution and it has attracted lots of attentions. However, existing approaches ignore a fact that people would label the pairs of code fragments as clone only if they happen to discover the clones while a huge number of undiscovered clone pairs and non-clone pairs are left unlabeled. In this paper, we argue that the clone detection task in the real-world should be formalized as a Positive-Unlabeled (PU) learning problem, and address this problem by proposing a novel positive and unlabeled learning approach, namely CDPU, to effectively detect software functional clones, i.e., pieces of codes with similar functionality but differing in both syntactical and lexical level, where adversarial training is employed to improve the robustness of the learned model to those non-clone pairs that look extremely similar but behave differently. Experiments on software clone detection benchmarks indicate that the proposed approach together with adversarial training outperforms the state-of-the-art approaches for software functional clone detection.

1 Introduction
Software clone are code fragments with similar functionalities, and they can be created by reusing code by copying, pasting and modifying [Roy and Cordy, 2007], or when an engineer unknowingly develops an implementation with similar functionality to an existing one [White et al., 2016]. Software clones introduce difficulties in software maintenance and cause bug propagation, and it has attracted lots of attentions.

Software functional clones are syntactically dissimilar code fragments that implement the same functionality. This kind of software clones is the most difficult to detect comparing with other clone types [Roy and Cordy, 2007; Wei and Li, 2017] since the code fragments can be quite different in both lexical and syntactical level and only similar in its functionality (e.g., summation implemented with for-loop and recursion). However, detecting software functional clone is very important not only because it takes up the majority among all clone types, but it is of great help in detecting plagiarism or copyright infringement [Baker, 1995; Brixel et al., 2010].

Many approaches have been proposed to detect software clone. They can be categorized into approaches using supervised information and approaches without it. Most of approaches use no supervised information, they express code fragments with features using only the inherent property of source code. For example, 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] exploits AST (Abstract Syntax Tree) to measure the structure similarity of two code fragments. In [White et al., 2016] the author proposed to learn latent features for source codes via autoencoder automatically [Socher et al., 2011]. Recently, CDLH [Wei and Li, 2017] has been proposed to formalize the software clone detection as a supervised learning to hash problem and learn supervised deep features in an end-to-end way for software functional clone detection. Clone detection approaches which use only inherent property of source code cannot effectively detect the functional similarity between code fragments, since two code fragments with similar functionality may be highly different in both lexical and syntactical level. Therefore, only approaches using supervised information can effectively detect software functional clone.

However, to the best of our knowledge, although formalizing clone detection as a supervised learning problem can effectively guide the feature learning process, usually it is difficult to obtain enough labeled data to train deep models, and the distribution of labeled data can be inconsistent with the underlying distribution. Since in practice the accumulation of labeled data is based on report of human labelers, which intend to report only clone pairs they happen to discover, and a huge number of undiscovered clone pairs and non-clone pairs are left unlabeled due to the limited labeling effort. The limited labeling effort results in limited labeled clone pairs, which leads to a serious problem: two code fragments that
look extremely similar may have different functionalities (as shown in Figure 1), while they are very easily to be mistaken as clone pairs since it is easy to find clone pairs that look similar in labeled data to mislead and there are no labeled non-clone pairs of such cases to help the clone detector discriminate against it. Therefore, a robust model is needed to detect following cases using only limited number of clone pairs:

- a) lexically and syntactically dissimilar code fragments with similar functionalities,
- b) lexically and syntactically similar code fragments with different functionalities.

In this paper, we argue that the clone detection task in the real-world should be formalized as a Positive-Unlabeled (PU) learning problem, and propose an effective approach for software functional clone detection named CDPU (software Clone Detection with Positive-Unlabeled learning) to leverage the unlabeled data to improve the detection performance, which also employs adversarial training mechanism [Goodfellow et al., 2015] to improve the robustness of the learned model to those non-clone pairs that look extremely similar but behave differently. Moreover, the detection process is quite efficient by using hash as the feature. Experiments on real-world clone detection benchmarks indicate that CDPU outperforms the state-of-the-art approaches for software functional clone detection.

This paper makes the following contributions:

- We first put forward that the clone detection task should respect to the way how and how many labels are collected, i.e., only limited number of clone pairs are discovered in practice. Consequently, it would be more natural to formalize the clone detection task as a PU learning problem.

- We propose a novel functional clone detection approach called CDPU which is able to leverage the unlabeled data to improve the detection performance. Such an approach is equipped with an adversarial training mechanism to further improve the robustness to non-clone pairs that look extremely similar but behave differently.

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

2 Problem Definition

Given $n$ code fragments $\{C_1, \ldots, C_n\}$ where $C_i$ is the $i$-th raw code fragment, and pairwise labels to indicate whether two code fragments belong to a clone pair: $y_{i,j} = 1$ if $(C_i, C_j)$ is a clone pair, $y_{i,j} = -1$ if $(C_i, C_j)$ is not a clone pair, and $y_{i,j} = 0$ if it has not been labeled. After applying a non-linear mapping function $\phi$, the raw code fragments are transformed into real-valued representations $z_i = \phi(C_i), \forall i \in [n]$, where $[n] = \{1, 2, \ldots, n\}$. Then the sets of positive, negative and unlabeled data are denoted as:

$$
D_p = \{(z_i, z_j) | i, j \in [n_p], i < j\}, \quad y_{i,j} = 1,
$$

$$
D_N = \{(z'_i, z'_j) | i, j \in [n_N], i < j\}, \quad y_{i,j} = -1,
$$

$$
D_U = \{(z_k, z_l) | k, l \in [n_U], k < l\}, \quad y_{k, l} = 0.
$$

where $n_p$ is the number of labeled positive data, $n_N$ is the number of labeled negative data, and $n_U$ is the number of unlabeled data. Since we formalize clone detection as a PU learning problem, the train data only consists of positive data $D_p$ and unlabeled data $D_U$, and our goal is to learn a function $\Phi$ which estimates whether two code fragments belong to a clone pair.

Specifically, we simultaneously learn the representation mapping function $\phi$ and a hash function $\psi : \mathbb{R}^d \rightarrow \{-1, 1\}^m$ mapping the $d$ dimensional representation into the Hamming space (i.e. $\psi(z_i) = [h_1(z_i), h_2(z_i), \ldots, h_m(z_i)], \forall i \in [n]$ encoding $\{z_i\}^n$ into binary hash codes $\{a_i^{(j)}\}$), so that the Hamming distance between the hash codes of two clone pairs can be as small as possible, and the distance between hash codes of none-clone pairs can be as large as possible. We use a function $S(z_i, z_j) = \frac{1}{m}\psi(z_i)\psi(z_j)^T$ to denote the similarity score of two code fragment representations. Given a pair of hash codes $\{a_i, a_j\}$, we apply a common function $g(a_i, a_j) = \mathbb{I}\left(\sum_{k=1}^m \frac{1}{4} * (a_{i,k} - a_{j,k})^2 \leq \text{thr}\right)$ 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 $\Phi(C_i, C_j) = g(\psi(\phi(C_i)), \psi(\phi(C_j)))$.

3 The Proposed Approach: CDPU

The overall framework of the CDPU is illustrated in Figure 2. We first transform the raw code fragments into digital features if 1

$$
1 Usually, the threshold $\text{thr}$ is set as 2.
It has been proved that PU-AUC risk $R_{PU}$ which is estimated by positive and unlabeled data, is equivalent to the supervised PN-AUC risk $R_{PN}$ with a linear transformation, where $R_{PU}$ and $R_{PN}$ are defined as [Xie and Li, 2018]:

$$R_{PU} = \mathbb{E}_{(z_i, z_j) \in D_P} \left[ \mathbb{E}_{(z_k, z_l) \in D_U} [l_{01}(S(z_i, z_j) - S(z_k, z_l))] \right]$$

$$R_{PN} = \mathbb{E}_{(z_i, z_j) \in D_P} \left[ \mathbb{E}_{(z_i', z_j') \in D_N} [l_{01}(S(z_i, z_j) - S(z_i', z_j'))] \right]$$

And the conclusion is:

$$R_{PU} = \frac{1}{2} \theta_P + \theta_N R_{PN}$$

The above conclusion is obtained mainly due to the observation that the unlabeled data comprises of $\theta_P$ percentage of positive data and $\theta_N$ percentage of negative data, where $\theta_P$ and $\theta_N$ are the prior probabilities of the positive and negative class of the whole dataset. Besides, the expected risk of pairs over $D_P \times D_P$ is symmetric, so the probability of ranking a labeled clone pair before an unlabeled clone pair is 1/2.

The above conclusion suggests that it is plausible to optimize $R_{PN}$ risk instead if we want to optimize the $R_{PU}$ risk. Specifically, under PU learning framework, clone detection task can be solved simply by treating the unlabeled data as negative data then optimizing the $R_{PN}$ risk, and it is unnecessary to guess the labels of unlabeled data or estimate the class prior beforehand.

Based on above claim, we define the optimization problem as follows:

$$\min_{\phi} L(Y, Z), \text{ where } L(Y, Z) = \frac{1}{n_{PU}} \sum_{(z_i, z_j) \in D_P} \sum_{(z_k, z_l) \in D_U} l(S(z_i, z_j) - S(z_k, z_l))$$

where $Z$ is the concatenation of all representations of code fragments, and $Y$ is the corresponding pairwise labels for these code fragments. Since the 0-1 loss is discrete and difficult to optimize, we replace it with the square loss $l(z) = (1 - z)^2$ in the experiment.

### 3.2 Feature Learning Process

Recently deep learning methods exhibit its superior performance in many applications, so we utilize deep learning methods as our model to extract effective representations for raw code fragments. Besides, as we hope that the detection process should be fast, we transform the real-valued representations into binary hash codes and use the hash codes as the final features to facilitate the detection process.

Specifically, the feature learning process contains two layers, the representation extraction layer and the hashing layer [Wei and Li, 2017]. The inputs are raw code fragments, they are first transformed into ASTs (Abstract Syntax Trees), then AST-based LSTM is applied to obtain the real-valued representation for each code fragment (i.e., representation extraction layer). After that, the hashing layer encodes those representations into binary hash codes, 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 binary hashcodes.

The first part is the representation extraction layer, which uses the AST-based LSTM to extract real-valued representations for code fragments. Unlike traditional LSTM [Zaremba and Sutskever, 2014] which processes expression in a chain way, AST-based LSTM processes the expression following the AST structure. Specifically, AST-based LSTM starts from the leaf nodes, and updates the parent nodes with information carried by hidden states of leaf nodes. Then information flows in a bottom-up way, from children to father. AST-based LSTM leverages the AST to capture structure information of code fragments and LSTM to extract the semantic information carried by lexical tokens of source codes, so it is able to incorporate both the lexical and syntactical information of source codes. An AST-based LSTM unit is updated as following:

$$i = \sigma(W_i x + \sum_{l=1}^{L} U_{il} r_l + b_i),$$

$$f_l = \sigma(W_{fl} x + U_{il} r_l + b_{fl}), \ l = 1, 2, \ldots, L,$$

$$o = \sigma(W_o x + \sum_{l=1}^{L} U_{ol} r_l + b_o),$$

$$u = \tanh(W_u x + \sum_{l=1}^{L} U_{ul} r_l + b_u),$$

$$c = i \odot u + \sum_{l=1}^{L} f_l \odot c_l,$$

$$z = o \odot \tanh(c),$$

where $x$ is the input word embedding of the corresponding token, $L$ is the number of children, $f_l$ $(l = 1, 2, \ldots, L)$ are $L$ forget gates for children of the AST node, $l$ is index number for its children, $W_i, W_{fl}, W_o, U_{il}, U_{ol}, U_{ul}$ are weight matrices, $b_i, b_{fl}, b_o, b_u$ are bias vectors, $\sigma$ is the logistic sigmoid function and $\odot$ is element-wise multiplication. Notice that each AST-based LSTM unit has $L$ forget gates.
each for its children nodes, and the hidden state of the root node $z_l$ is considered as the representation of this expression.

After obtaining real-valued representations with AST-based LSTM, the second part is the hashing layer which transforms these real-valued representations into binary hash codes to facilitate the detection process:

$$a_i = \psi(z_i) = \text{sign}(W_h^T z_i + b_h), \forall i \in [n]$$ \hspace{1cm} (6)

where $W_h, b_h$ are parameters for the hash function. Then the obtained hashcodes are the final deep features representing code fragments.

### 3.3 The Training Process

From the feature learning process it can be obtained that non-clone pairs that look extremely similar but behave differently will be mapped to similar features, and they are very likely to be detected as clone pairs, which cannot handle cases like that in Figure 1. Therefore, we apply adversarial training [Goodfellow et al., 2014; Miyato et al., 2016] to get a robust clone detector.

Adversarial training is a regularization method to help improve the robustness of learners to small, approximately worst case perturbations (as described in Figure 1). In clone detection scenario, we train the classifier to be robust to perturbations of the extracted representations of code fragments. At each step of training, the worst case perturbations are generated by maximizing the loss function as follows:

$$R = \arg\max_R L(Y, Z + R)$$

then the learners are trained to be robust to these worst case perturbations through minimizing loss function.

Since exactly minimizing above equation is intractable for many models such as neural networks, we approximate this value by linearizing $L$ around $Z$ as suggested in [Goodfellow et al., 2014]. With the linear approximation and a $L_2$ norm constraint the resulting perturbation is:

$$R = \epsilon g/\|g\|_2, \text{where } g = \nabla_Z L(Y, Z)$$ \hspace{1cm} (7)

where $\epsilon$ is a small constant.

To be robust to the adversarial perturbation defined in Equation 7, we define the adversarial loss by

$$L_{adv}(Y, Z + R) = \frac{1}{n_P n_U} \sum_{(z_i, z_j) \in D_P} \sum_{(z_k, z_l) \in D_U} l(S(z_i + r_i, z_j + r_j) - S(z_k + r_k, z_l + r_l))$$ \hspace{1cm} (8)

By minimizing Equation 8 the clone detector is trained to be robust to small perturbations.

### 4 Experiment

In this section, we conduct experiments on real-world datasets to verify the effectiveness of our proposed CDPU. Specifically, first we introduce the experimental setting including the datasets used and our modifications on the data to make it suit our setting. Then we compare CDPU with several state-of-the-art clone detection approaches together with advanced approaches from PU learning category. After that we show the improvement in performance of a robust model by using adversarial training. Finally we study the performance variations with the number of the labeled clone pairs.

### 4.1 Experimental Setting

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

BigCloneBench consists of projects from 25,000 systems, covers 10 functionalities including 6,000,000 clone pairs and 260,000 non-clone pairs. All labeled clone types are given by domain experts. We discard code fragments without any tagged true or false clone pairs, and use the remaining 9,134 code fragments. Note that in practice it is nontrivial to label so many clone pairs for clone detection task. Therefore, we will remove labels for a large percentage of labeled clone pairs and all labeled non-clone pairs, only keep a small amount of clone pairs to simulate the real-world labeling behavior.

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. In the experiment, we select the first 15 programming problems, and for each problem there are 500 source code files. For OJClone, we do not have experts to distinguish different clone types. Note that there are only positive labels for OJClone.

For BigCloneBench, a code fragment is a method, and for OJClone a code fragment is a file. In order to construct AST structure, we use javalang, a pure python library for working with Java source code, to parse JAVA code to ASTs, and apply pycparser to parse C files to ASTs. To obtain word embeddings for tokens of original code fragments, we use word2vec to generate word embeddings of length 100 for both datasets. We use hash code length of 32 in the experiment, and other hash code lengths cause similar results.

In practice, we hope that we could label as few data as possible. Therefore, in the experiment we randomly keep at most 2 clone pairs labeled for each code fragment and treat the rest clone pairs as unlabeled data. In later experiments, we will study the influence of the number of labeled clone pairs on the performance. The overall information of datasets are listed in Table 1.

We use precision (P), recall (R), F1 value, and F1 with respect to various clone types as performance measurement. These clone types including [Roy and Cordy, 2007; Wei and

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Language</th>
<th># code fragments</th>
<th>AVG length</th>
<th>% data labeled</th>
</tr>
</thead>
<tbody>
<tr>
<td>BigCloneBench</td>
<td>JAVA</td>
<td>9,134</td>
<td>28.60</td>
<td>0.021</td>
</tr>
<tr>
<td>OJClone</td>
<td>C</td>
<td>7,500</td>
<td>35.25</td>
<td>0.026</td>
</tr>
</tbody>
</table>

Table 1: Overall information of datasets.
Table 2: Precision, recall and F1 comparison of all clone detection approaches.

<table>
<thead>
<tr>
<th>Approaches</th>
<th>BigCloneBench</th>
<th>OJClone</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>R</td>
</tr>
<tr>
<td>Deckard</td>
<td>0.93</td>
<td>0.02</td>
</tr>
<tr>
<td>DLC</td>
<td>0.95</td>
<td>0.01</td>
</tr>
<tr>
<td>SourcererCC</td>
<td>0.88</td>
<td>0.02</td>
</tr>
<tr>
<td>CD-H</td>
<td>0.70</td>
<td>0.01</td>
</tr>
<tr>
<td>CDLH</td>
<td>0.81</td>
<td>0.21</td>
</tr>
<tr>
<td>CDPU</td>
<td>0.52</td>
<td>0.50</td>
</tr>
</tbody>
</table>

Table 3: Percentage of various clone types for BigCloneBench. T1, T2, ST3, MT3 and T4 mean Type-1, Type-2, Strong Type-3, Mid Type-3 and Type-4 clone.

<table>
<thead>
<tr>
<th>Approaches</th>
<th>T1</th>
<th>T2</th>
<th>ST3</th>
<th>MT3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deckard</td>
<td>0.73</td>
<td>0.71</td>
<td>0.54</td>
<td>0.21</td>
<td>0.02</td>
</tr>
<tr>
<td>SourcererCC</td>
<td>0.94</td>
<td>0.93</td>
<td>0.77</td>
<td>0.10</td>
<td>0.00</td>
</tr>
<tr>
<td>CD-H</td>
<td>1.00</td>
<td>0.87</td>
<td>0.73</td>
<td>0.05</td>
<td>0.01</td>
</tr>
<tr>
<td>CDLH</td>
<td>1.00</td>
<td>0.91</td>
<td>0.61</td>
<td>0.32</td>
<td></td>
</tr>
<tr>
<td>CDPU</td>
<td>1.00</td>
<td>1.00</td>
<td>0.98</td>
<td>0.70</td>
<td>0.51</td>
</tr>
</tbody>
</table>

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.

4.2 Performance Comparison

In this section, we compare our proposed CDPU with several state-of-the-art clone detection approaches to verify its effectiveness as a clone detection tool:

- 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.

Apart from the above approaches, we also compare following approaches with CDPU:

- CDLH [Wei and Li, 2017], which formalizes the software clone detection as a supervised learning problem and learns supervised deep features in an end-to-end way for software functional clone detection.
- C-DH [du Plessis et al., 2015], an advanced PU learning approach, which is proved to converge to the optimal solutions as a convex formulation using proposed double hinge loss.

We use CDLH to validate that it is enough to optimize rank loss by treating all unlabeled data as negative. As there are no negative data for CDLH to train a supervised model, we randomly sample similar amount of unlabeled data as negative data together with the labeled positive data for training. For C-DH, we use the feature learning process of CDPU to provide representations for it, and replace the loss function of CDPU to double hinge loss to learn in an end-to-end way.

Table 2 tabulates the precision, recall and F1 values of various approaches. It can be observed that CDPU outperforms other approaches in terms of F1 values. Approaches using supervised information such as CDLH and CDPU perform better than others, since they can learn patterns from code fragments with similar functionality guided by supervised information while others cannot. Thus their recall value is higher than unsupervised approaches due to their ability to detect software functional clone, which takes up more than 98% (T4) over all clone types for dataset BigCloneBench according to Table 3. Moreover, these approaches using the same deep learning model which can incorporate both the lexical and syntactical information of source code. Among these supervised approaches, there need to be more positive data for C-DH to learn a good model, and the estimated class prior may be misleading for later training process. CDLH can only use supervised data for training, yet there are not enough labeled data for it since we only have small amount of labeled positive data, which affects its performance. CDPU is able to optimize unbiased rank loss using all data by treating unlabeled data as negative, thus it outperforms other supervised approaches, which verifies the effectiveness of the rank loss.

As for dataset OJClone, most of its clone pairs 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. Therefore the recall for unsupervised approaches is inferior compared with their precision value. 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 overlapped tokens such as a, b, i, j, it is also unsuitable to treat them as clones.

Table 4 tabulates F1 values with respect to various clone types. These five fine-grained categories are proposed by [Svajlenko et al., 2014], they divide clone types based on code similarity: 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). As only dataset BigCloneBench is tagged with these five fine-grained categories, we only show these results for BigCloneBench. From the results we can see that all approaches achieve good performance for Type-1 and Type-2 clones, since these two clone types are relatively easier to detect
by simply comparing the tokens or syntactical structures of
codes. However, the detection performance degrades asym-
ptotically for Type-3 and Type-4 clones, especially for Type-4
due to the improvement of detection difficulty. For unsup-
ervised approaches, they can hardly detect any Type-4 clones,
which causes their relatively low overall F1 values. Appro-
aches using supervised information perform better, espe-
cially for CDPU, which outperforms other approaches. This
validates the effectiveness of CDPU in detecting software
functional clones.

4.3 Effectiveness of Adversarial Training
In this section, to validate that a robust model using adver-
sarial training outperforms model without it, we compare
the performance of a model using adversarial training with model
that does not use it. We name the approach using the rank loss
and feature learning process of CDPU but without adversarial
training as CDPU\(^{-}\). We compare the performance of CDPU
with CDPU\(^{-}\) to show the effectiveness of adversarial train-
ing.

Table 5 tabulates the precision, recall and F1 values of
CDPU\(^{-}\) and CDPU. From the results it is clear that CDPU
outperforms CDPU\(^{-}\). CDPU is robust to small perturbations,
so for two code fragments with minor differences while dif-
ferent functionalities, CDPU can differentiate from them, yet
CDPU\(^{-}\) cannot. Therefore, the precision value of CDPU is
higher than CDPU\(^{-}\), which leads to better performance of
CDPU.

Table 6 tabulates F1 values comparison with respect to
various clone types on BigCloneBench between CDPU and
CDPU\(^{-}\). From the table we can see that for Type-1, Type-
2 and Strong Type-3 clone types, both CDPU and CDPU\(^{-}\)
achieve good performance. These clone types are relatively
easy to detect with lexical or syntactical information of source
codes, for which it is enough to use the feature learning pro-
cess of CDPU shared by CDPU and CDPU\(^{-}\). While for Mid
Type-3 and software functional clones CDPU outperforms
CDPU\(^{-}\), especially for software functional clone. The
results on these relatively difficult clone types reveal the benefit
of robust model using adversarial training.

![Figure 3: F1 values of the compared approaches with respect to the
number of code fragments labeled as clone for each code fragment
\(n_{\text{labeled}}\).](image)

Figure 3: F1 values of the compared approaches with respect to the
number of code fragments labeled as clone for each code fragment
\(n_{\text{labeled}}\).

4.4 Influence of the Number of the Labeled Clone
Pairs
In this section, we study the influence of the number of la-
beled clone pairs has on the performance. Specifically, we
denote the number of code fragments labeled as clone for
each code fragment as \(n_{\text{labeled}}\), and show the tendency of per-
formance variation with respect to \(n_{\text{labeled}}\) to reveal that how
large should \(n_{\text{labeled}}\) be to obtain comparable performance
with model using all labeled information.

Figure 3 depicts the performance of CDPU and CDLH in
terms of F1 values. CDLH which leverages all labeled data is
considered as the upper bound of the performance of CDPU
which uses only a tiny portion of labeled data. It can be ob-
erved from the Figure that as \(n_{\text{labeled}}\) increases, the perform-
ance of CDPU gradually approaches CDLH. For example,
when \(n_{\text{labeled}}\) reaches 20 (which is roughly 3.04\% of the en-
tire labeled data), the performance of CDPU is very close to
CDLH. This suggests that as \(n_{\text{labeled}}\) increases, the proposed
CDPU can gradually converge to the performance of CDLH,
which requires the entire labeled data. Similar trends can be
observed on dataset OJClone.

5 Conclusion
Successful functional clone detection requires lots of labeled
data to train a well-performed model, and the distribution of
labeled data should be unbiased compared with the underly-
ing real-world distribution. However, in reality people would
only label clone pairs that they happen to discover, and a
large proportion of undiscovered clone pairs and non-clone
pairs are left unlabeled. To learn a well-performed model un-
der this circumstance, in this paper we argue that software
code detection in the real-world should be formalized as a
PU learning task, and propose a novel PU learning model
called CDPU, which can leverage the unlabeled data to im-
prove the performance of functional clone detection, equip-
ning with special adversarial training mechanism to improve
the robustness of trained model to those non-clone pairs that
look extremely similar but behave differently. Experimental
results indicate that CDPU can leverage the unlabeled data to
improve the detection performance, and adversarial training
can actually help to further improve the robustness of clone
detector. New models that can extract more information from
the code will be considered in the future.
Acknowledgements
This research was supported by National Key Research and Development Program (2017YFB1001903) and NSFC (61422304).

References


