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.

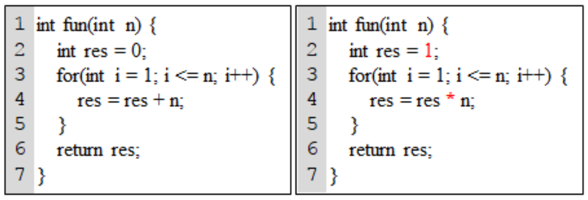
1 Introduction

Software clones 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; Brixtel *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



```
1 int fun(int n) {
2     int res = 0;
3     for(int i = 1; i <= n; i++) {
4         res = res + n;
5     }
6     return res;
7 }
```

```
1 int fun(int n) {
2     int res = 1;
3     for(int i = 1; i <= n; i++) {
4         res = res * n;
5     }
6     return res;
7 }
```

Figure 1: Small differences can change the functionality of a code fragment (code in the right only differs in '1' in Line 2 and '*' in Line 4 from code in the left).

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 them 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, and 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 hashcode 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 $\{\mathrm{C}_1, \cdots, \mathrm{C}_n\}$ where C_i is the i-th raw code fragment, and pairwise labels to indicate whether two code fragments belong to a clone pair: $y_{i,j} = 1$ if $(\mathrm{C}_i, \mathrm{C}_j)$ is a clone pair, $y_{i,j} = -1$ if $(\mathrm{C}_i, \mathrm{C}_j)$ is a not a clone pair, and $y_{i,j} = 0$ if it has not been labeled. After applying a non-linear mapping function ϕ, the raw code fragments are transformed into real-valued representations $\boldsymbol{z}_i = \phi(\mathrm{C}_i), \forall i \in [n]$, where $[n] = \{1, 2, \cdots, n\}$. Then the sets of positive, negative and unlabeled data are denoted as:

$$\mathcal{D}_P = \{(\boldsymbol{z}_i, \boldsymbol{z}_j) | i, j \in [n_P], i < j\}, \text{where } y_{i,j} = 1,$$
$$\mathcal{D}_N = \{(\boldsymbol{z}'_i, \boldsymbol{z}'_j) | i, j \in [n_N], i < j\}, \text{where } y_{i,j} = -1, \text{and}$$
$$\mathcal{D}_U = \{(\boldsymbol{z}_k, \boldsymbol{z}_l) | k, l \in [n_U], k < l\}, \text{where } y_{k,l} = 0.$$

where n_P is the number of labeled positive data, n_N is the number of labeled negative data, n_U is the number of unlabeled data. Since we formalize clone detection task as a PU learning problem, the train data only consists of positive data $\mathcal{D}_P$ and unlabeled data $\mathcal{D}_U$, and our goal is to learn a function Φ which estimates whether two code fragments belong to a clone pair.

Specifically, we simultaneously learn the representation mapping function ϕ and a hash function $\psi : \mathbb{R}^d \to \{-1, 1\}^m$ 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 hash codes of none-clone pairs can be as large as possible. We use a function $S(\boldsymbol{z}_i, \boldsymbol{z}_j) = \frac{1}{m}\psi(\boldsymbol{z}_i)\psi(\boldsymbol{z}_j)^T$ to denote the similarity score of two code fragment representations. 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: CDPU

The overall framework of the CDPU is illustrated in Figure 2. we first transform the raw code fragments into digital features based on a deep model whose structure is shown in Section 3.2, then we utilized adversarial training by adding perturbations to extracted features to learn a robust model (Section 3.3). The whole process is guided by PU-AUC risk defined in Section 3.1.

3.1 Problem Formulation

Existing Positive-Unlabeled learning approaches usually guess labels for the unlabeled data or presume that the class prior is known beforehand [Liu *et al.*, 2002; du Plessis *et al.*, 2015; Sakai *et al.*, 2017]. However, the class prior is hard to know with only positive labeled data and the estimated class prior may be misleading for later computation. Actually, it is unnecessary to guess the labels for unlabeled data or estimate the class prior if we optimize the rank loss [Xie and Li, 2018]. It has been shown that unbiased PU-AUC optimization is achievable without distributional assumptions or prior knowledge about the distribution or class prior probabilities.

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

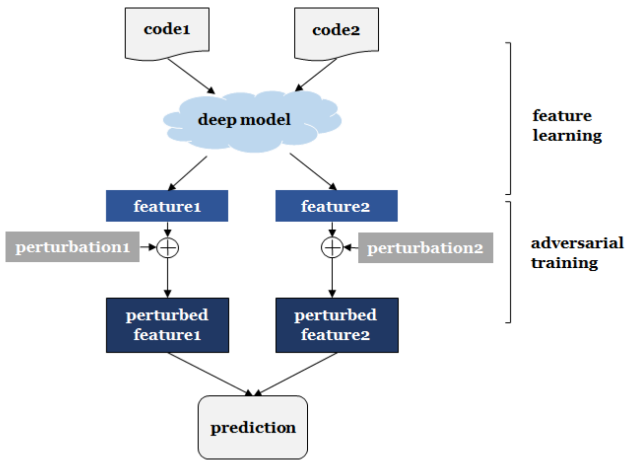


Figure 2: The overall framework of CDPU.

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]:

$$R_{PU} = \underset{(\boldsymbol{z}_i,\boldsymbol{z}_j)\in\mathcal{D}_P}{\mathbb{E}}[\underset{(\boldsymbol{z}_k,\boldsymbol{z}_l)\in\mathcal{D}_U}{\mathbb{E}}[l_{01}(S(\boldsymbol{z}_i,\boldsymbol{z}_j)-S(\boldsymbol{z}_k,\boldsymbol{z}_l))]] \tag{1}$$

$$R_{PN} = \underset{(\boldsymbol{z}_i,\boldsymbol{z}_j)\in\mathcal{D}_P}{\mathbb{E}}[\underset{(\boldsymbol{z}'_i,\boldsymbol{z}'_j)\in\mathcal{D}_N}{\mathbb{E}}[l_{01}(S(\boldsymbol{z}_i,\boldsymbol{z}_j)-S(\boldsymbol{z}'_i,\boldsymbol{z}'_j))]] \tag{2}$$

And the conclusion is:

$$R_{PU} = \frac{1}{2}\theta_P + \theta_N R_{PN} \tag{3}$$

The above conclusion is obtained mainly due to the observation that the unlabeled data comprises of θ_P percentage of positive data and θ_N percentage of negative data, where θ_P and θ_N are the prior probabilities of the positive and negative class of the whole dataset. Besides, the expected risk of pairs over $\mathcal{D}_P \times \mathcal{D}_P$ 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:

$$\min_{\phi,\psi} L(Y,Z), \text{where } L(Y,Z) = \frac{1}{n_P n_U}\sum_{(\boldsymbol{z}_i,\boldsymbol{z}_j)\in\mathcal{D}_P}\sum_{(\boldsymbol{z}_k,\boldsymbol{z}_l)\in\mathcal{D}_U} l(S(\boldsymbol{z}_i,\boldsymbol{z}_j)-S(\boldsymbol{z}_k,\boldsymbol{z}_l)) \tag{4}$$

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:

$$\begin{aligned}
\boldsymbol{i} &= \sigma(W_i\boldsymbol{x} + \sum_{l=1}^{L} U_{il}\boldsymbol{r}_l + \boldsymbol{b}_i),\\
\boldsymbol{f}_l &= \sigma(W_f\boldsymbol{x} + U_{fl}\boldsymbol{r}_l + \boldsymbol{b}_f),\ l = 1,2,\ldots,L\\
\boldsymbol{o} &= \sigma(W_o\boldsymbol{x} + \sum_{l=1}^{L} U_{ol}\boldsymbol{r}_l + \boldsymbol{b}_o),\\
\boldsymbol{u} &= \tanh(W_u\boldsymbol{x} + \sum_{l=1}^{L} U_{ul}\boldsymbol{r}_l + \boldsymbol{b}_u),\\
\boldsymbol{c} &= \boldsymbol{i}\odot\boldsymbol{u} + \sum_{l=1}^{L}\boldsymbol{f}_l\odot\boldsymbol{c}_l,\\
\boldsymbol{z} &= \boldsymbol{o}\odot\tanh(\boldsymbol{c}),
\end{aligned} \tag{5}$$

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 each AST-based LSTM unit has L forget gates each for its children nodes, and the hidden state of the root node $\boldsymbol{z_i}$ 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:

$$\boldsymbol{a}_i = \psi(\boldsymbol{z}_i) = \operatorname{sign}(W_h^T \boldsymbol{z}_i + \boldsymbol{b}_h), \forall i \in [n] \tag{6}$$

where $W_h, \boldsymbol{b}_h$ 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:

$$R = \underset{R}{argmax}\, L(Y, Z + R)$$

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:

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

where ϵ is a small constant.

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

$$L_{adv}(Y, Z + R) = \frac{1}{n_P n_U} \sum_{(\boldsymbol{z}_i,\boldsymbol{z}_j)\in\mathcal{D}_P} \sum_{(\boldsymbol{z}_k,\boldsymbol{z}_l)\in\mathcal{D}_U} l(S(\boldsymbol{z}_i + \boldsymbol{r}_i, \boldsymbol{z}_j + \boldsymbol{r}_j) - S(\boldsymbol{z}_k + \boldsymbol{r}_k, \boldsymbol{z}_l + \boldsymbol{r}_l)) \tag{8}$$

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.

Datasets	Language	# code fragments	AVG length	% data labeled
BigCloneBench	JAVA	9,134	28.60	0.021
OJClone	C	7,500	35.25	0.026

Table 1: Overall information of datasets.

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 [2](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[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. 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 has 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

[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

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
C-DH	0.70	0.01	0.01	0.09	0.23	0.13
CDLH	0.81	0.21	0.33	0.12	0.18	0.15
CDPU	0.52	0.50	0.51	0.19	0.17	0.18

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

Li, 2017]: Type-1: identical code fragments in addition to variations in comments and layout; Type-2: identical code fragments in addition to differences in identifier names and literal values; Type-3: syntactically similar code that differ at the statement level; Type-4: software functional clone. We use 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 software functional clone.

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.

T1	T2	ST3	MT3	T4
0.50%	0.10%	0.20%	1.00%	98.20%

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.

Approaches	T1	T2	ST3	MT3	T4
Deckard	0.73	0.71	0.54	0.21	0.02
DLC	**1.00**	0.97	0.60	0.03	0.00
SourcererCC	0.94	0.93	0.77	0.10	0.00
C-DH	**1.00**	0.87	0.73	0.05	0.01
CDLH	**1.00**	**1.00**	0.91	0.61	0.32
CDPU	**1.00**	**1.00**	**0.98**	**0.70**	**0.51**

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.

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 various clone types, 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 asymptotically for Type-3 and Type-4 clones, especially for Type-4 due to the improvement of detection difficulty. For unsupervised approaches, they can hardly detect any Type-4 clones, which causes their relatively low overall F1 values. Approaches using supervised information perform better, especially 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 adversarial 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 training.

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 different 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 process 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.

Approaches	BigCloneBench			OJClone		
	P	R	F1	P	R	F1
CDPU$^-$	0.44	0.45	0.44	0.16	0.13	0.14
CDPU	**0.52**	**0.50**	**0.51**	**0.19**	**0.17**	**0.18**

Table 5: Precision, recall and F1 comparison of approach using adversarial training with approach without it.

Approaches	T1	T2	ST3	MT3	T4
CDPU$^-$	**1.00**	**1.00**	**0.98**	0.66	0.43
CDPU	**1.00**	**1.00**	**0.98**	**0.70**	**0.51**

Table 6: Comparison of F1 values with respect to various clone types on BigCloneBench between approach using adversarial training and approach without it, the best performed across each clone type is emphasized with boldface.

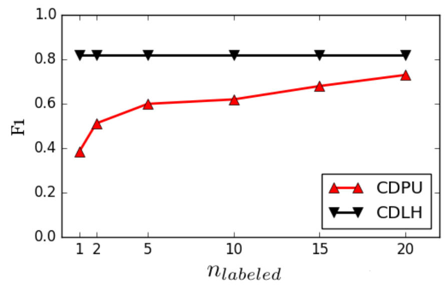


Figure 3: F1 values of the compared approaches with respect to the number of code fragments labeled as clone for each code fragment $n_{labeled}$.

4.4 Influence of the Number of the Labeled Clone Pairs

In this section, we study the influence of the number of labeled clone pairs has on the performance. Specifically, we denote the number of code fragments labeled as clone for each code fragment as $n_{labeled}$, and show the tendency of performance variation with respect to $n_{labeled}$ to reveal that how large should $n_{labeled}$ 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 observed from the Figure that as $n_{labeled}$ increases, the performance of CDPU gradually approaches CDLH. For example, when $n_{labeled}$ reaches 20 (which is roughly 3.04% of the entire labeled data), the performance of CDPU is very close to CDLH. This suggests that as $n_{labeled}$ 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 underlying 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 under this circumstance, in this paper we argue that software clone 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 improve the performance of functional clone detection, equipping 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

[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.

[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.

[du Plessis *et al.*, 2015] Marthinus Christoffel du Plessis, Gang Niu, and Masashi Sugiyama. Convex formulation for learning from positive and unlabeled data. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 1386–1394, Lille, France, 2015.

[Goodfellow *et al.*, 2014] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *CoRR*, abs/1412.6572, 2014.

[Goodfellow *et al.*, 2015] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *International conference on learning representations*, 2015.

[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.

[Liu *et al.*, 2002] Bing Liu, Wee Sun Lee, Philip S. Yu, and Xiaoli Li. Partially supervised classification of text documents. In *Proceedings of the 19th International Conference on Machine Learning*, pages 387–394, Sydney, Australia, 2002.

[Miyato *et al.*, 2016] Takeru Miyato, Andrew M Dai, and Ian J Goodfellow. Adversarial training methods for semi-supervised text classification. *arXiv: Machine Learning*, 1, 2016.

[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 13th 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. *Queen's 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.

[Sakai *et al.*, 2017] Tomoya Sakai, Marthinus Christoffel du Plessis, Gang Niu, and Masashi Sugiyama. Semi-supervised classification based on classification from positive and unlabeled data. In *Proceedings of the 34th International Conference on Machine Learning*, pages 2998–3006, Sydney, NSW, Australia, 2017.

[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 *30th IEEE International Conference on Software Maintenance and Evolution*, pages 476–480, Victoria, BC, Canada, 2014.

[Wei and Li, 2017] Huihui Wei and Ming Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 3034–3040, Melbourne, Australia, 2017.

[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.

[Xie and Li, 2018] Zheng Xie and Ming Li. Semi-supervised auc optimization without guessing labels of unlabeled data. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*, New Orleans, LA, 2018.

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