Can Neural Clone Detection Generalize to Unseen Functionalities?

Chenyao Liu*
School of Software
Tsinghua University
liucy19@mails.tsinghua.edu.cn

Zeqi Lin§
Microsoft Research Asia
Zeqi.Lin@microsoft.com

Jian-Guang Lou
Microsoft Research Asia
jlou@microsoft.com

Lijie Wen§
School of Software
Tsinghua University
wenlj@tsinghua.edu.cn

Dongmei Zhang
Microsoft Research Asia
dongmeiz@microsoft.com

***Abstract*—Many recently proposed code clone detectors exploit neural networks to capture latent semantics of source code, thus achieving impressive results for detecting semantic clones. These neural clone detectors rely on the availability of large amounts of labeled training data. We identify a key oversight in the current evaluation methodology for neural clone detection: cross-functionality generalization (i.e., detecting semantic clones of which the functionalities are unseen in training). Specifically, we focus on this question: do neural clone detectors truly learn the ability to detect semantic clones, or they just learn how to model specific functionalities in training data while cannot generalize to realistic unseen functionalities? This paper investigates how the generalizability can be evaluated and improved.**

Our contributions are 3-folds: (1) We propose an evaluation methodology that can systematically measure the cross-functionality generalizability of neural clone detection. Based on this evaluation methodology, an empirical study is conducted and the results indicate that current neural clone detectors cannot generalize well as expected. (2) We conduct empirical analysis to understand key factors that can impact the generalizability. We investigate 3 factors: training data diversity, vocabulary, and locality. Results show that the performance loss on unseen functionalities can be reduced through addressing the out-of-vocabulary problem and increasing training data diversity. (3) We propose a human-in-the-loop mechanism that help adapt neural clone detectors to new code repositories containing lots of unseen functionalities. It improves annotation efficiency with the combination of transfer learning and active learning. Experimental results show that it reduces the amount of annotations by about 88%. Our code and data are publicly available[1].

***Index Terms*—Code Clone Detection, Generalization, Neural Network, Evaluation Methodology, Human-in-the-Loop**

I. Introduction

Code clone detection is the task of finding similar code fragment pairs (i.e., clones) within or between software systems. It has become an important part in many software engineering tasks, such as software refactoring ([1]–[4]), quality management ([5]–[8]), defect prediction [9], plagiarism detection ([9], [10]), and program comprehension ([9], [11]).

*Work done during an internship at Microsoft Research
§Corresponding author
[1]https://github.com/thousfeet/Functionality-generalization

In recent years, many neural network-based methods are proposed for detecting *semantic clones*, and they have achieved impressive results ([12]–[19]). Semantic clones are clones in which code fragments implement the same functionality, but may have low syntactic similarity. For example, a quick sort code and a heap sort code should be considered semantically equivalent. Traditional matching-based code clone detectors (e.g., token matching-based methods, tree matching-based methods, and graph matching-based methods work well in detecting syntactic clones, while previous studies ([20], [21]) found that they had limited success with semantic clones. To address this problem, a recent research trend is to leverage deep neural networks to effectively capture complex semantic information in code fragments. For example, some studies (e.g., CDLH [12], ASTNN [15], and TBCCD [16]) focus on learning from Abstract Syntax Trees (ASTs), and some other studies ([13], [19]) focus on learning from Control Flow Graphs (CFGs) or Program Dependency Graphs (PDGs). These studies have achieved impressive results: in widely-used code benchmarks for code clone detection (e.g., BigCloneBench [22], GCJ [13], and OJClone [23]), state-of-the-art neural clone detectors achieve more than 90% precision and recall.

Existing neural clone detectors are supervised, relying on a large number of annotated true/false code fragment pairs for training. This paper identifies a key oversight in the current evaluation methodology for neural clone detection:

Cross-Functionality Generalizability: the ability to detect semantic clones of which the functionalities have never been previously observed in the training dataset.

For example, a good neural clone detector should be able to find clones of sort algorithms, even if the training data contain no code fragments of sorting. This generalizability is a critical aspect to measure whether neural clone detection can be applied in practice at scale, because: (1) there are a potentially infinite number of functionalities in real-world software systems (especially for domain-specific software systems), making it almost impossible to construct a large-scale training dataset that covers most functionalities; (2) it is expensive and not scalable to annotate specific training dataset for each domain.

The current evaluation methodology for neural clone detection does not systematically test the cross-functionality generalizability. Due to annotation difficulties, the benchmarks with many semantic clones usually have a limited number of functionalities. For example, BigCloneBench, GCJ and OJClone have 43/12/104 functionalities respectively. Training/test sets are randomly sampled from all annotated code fragment pairs, with the restriction that a code fragment should not appear in both training set and test set. In this setting, whether a clone detector can generalize to unseen code fragments or not is well tested, but the cross-functionality generalizability is not. A reasonable concern is: do neural clone detectors really learn to model semantic equivalence of code fragment pairs, or they just simply remember fixed patterns of fixed functionalities?

In this paper, we aim to answer three questions:

1) How does neural clone detection generalize to unseen functionalities?
2) If the cross-functionality generalizability of neural clone detection is limited, what are the key factors that impact it?
3) In a new domain, how to learn an accurate neural clone detector with minimal cost?

Our first contribution is **a simple yet realistic evaluation methodology for the generalizability of neural clone detection and an empirical study based on it.** Currently, annotated code fragment pairs are divided into training/test sets based on code fragments. This setting tests whether the detector can generalize to unseen code fragments, but not a complete test of the generalizability for unseen functionalities. To address this problem, we improve the evaluation methodology through functionality-based re-partition of training/test sets. We then use this methodology to test the generalizability of neural clone detection on OJClone dataset (which contains 104 functionalities). In particular, we divide these 104 functionalities into 7 groups and run experiments on different training-test settings. Our empirical observation is: neural clone detection suffers from significant performance degradation (on average, F1 score decreases from 0.96 to 0.44) in cross-functionality settings.

This result motivates our second contribution: a series of experiments are conducted to **understand the factors that impact the cross-functionality generalizability.** Specifically, we examine 3 potential improvement directions:

1) **Training Data Diversity**. We define training data diversity as the total number of functionalities in training set. We hyphothesize that the cross-functionality generalizability can benefit from increasing training data diversity. Our empirical results confirm this hypothesis while significant marginal effects are observed.
2) **Vocabulary**. Neural networks in software engineering tasks usually suffer from the out-of-vocabulary (OOV) problem ([16], [24], [25]): tokens in test set may rarely or even never occur in training set, thus these tokens are not effectively modeled in neural networks. The vocabulary of different functionalities is likely to be very different. Therefore, we hypothesize that adressing the OOV problem can alleviate the lack of cross-functionality generalizability. Our empirical results confirm this hypothesis.
3) **Locality**. Common wisdom in machine learning community suggests that we should use attention mechanism to better model local structures of data, especially for their latent alignments ([26]–[29]). We apply this idea to neural clone detection and study whether it can help improve cross-functionality generalizability. Our empirical results show that this mechanism brings little improvement for generalizability.

Finally, our third contribution is **a human-in-the-loop mechanism that efficiently bootstraps neural clone detectors for unseen functionalities with only a small amount of human efforts.** The scenario is: we have a training dataset A, but we want to learn a good neural clone detector that can find semantic clones in a new, lower-resourced domain B (containing many functionalities that have never been previously observed in A). The key point of this human-in-the-loop mechanism is the combination of transfer learning and active learning: we learn a preliminary neural clone detector on domain A, then use it to actively select informative code fragment pairs in domain B for human annotation, thus transferring the neural clone detector from domain A to domain B. Experimental results show that this human-in-the-loop mechanism reduces the amount of annotations by 88%, thus alleviating the difficulty that neural clone detectors cannot well extend to various real-world code repositories.

More broadly, these contributions may impact research on neural source code representation (i.e., encoding code fragments to continuous vectors, based on neural networks), which has attracted much attention in recent years [30]. Neural source code representation has been widely used and achieved impressive results not only in code clone detection, but also in various software engineering tasks (e.g., code completion ([24], [25], [31]–[35]), code search ([36]–[39]) , code summarization ([40]–[48]), code translation ([49], [50]), and defect prediction ([51]–[53])). In this paper, we use code clone detection as a case study to show the importance of (1) studying the dependence of supervised neural methods on training data; (2) probing whether these methods can generalize beyond training data. We suggest that a generalization-aware evaluation methodology should be used to better evaluate neural methods in software engineering community, and more future efforts should be made to improve generalizability.

II. Background

A. Semantic Clones and Neural Clone Detection

Existing research work divides code clones into four major types ([54], [55]):

- **Type-1**: Identical code fragments, except for differences in white-space, layout and comments.

- **Type-2**: Identical code fragments, except for differences in identifier names and literal values, as well as Type-1 differences.
- **Type-3**: Syntactically similar code fragments that differ at the statement level. The fragments have statements added, modified and/or removed with respect to each other, in addition to Type-1 and Type-2 clone differences.
- **Type-4**: Syntactically dissimilar code fragments that implement the same functionality.

As there is no clear boundary between Type-3 clones and Type-4 clones, we vaguely define *semantic clones* as the union of Type-4 clones and Type-3 clones that cannot easily be detected by pre-defined rules.

Figure 1 shows an examlple of semantic clone (Type-4). Even through the functionality of these two code fragments is very simple (calculates x raised to the power n), different programmers may implement it in totally different ways.

```c
double power(double x, int n) {
    double temp;
    if(n==0) return 1.00;
    temp=power(x, n/2);
    if(n%2==0)
        return temp*temp;
    else {
        if(n>0) return x*temp*temp;
        else return (temp*temp)/x;
    }
}

double exponent(double x, int n) {
    if(n < 0 && n != INT_MIN) n--;
    double ans = 1.0;
    for(int i = 0; i < sizeof(int)*CHAR_BIT-1; i++) {
        if((n & 1) ^ (n < 0 && n != INT_MIN))
            ans *= x;
        x = x * x;
        n = n >> 1;
    }
    if(n == INT_MIN) ans *= x;
    return n < 0 ? 1.0/ans : ans;
}
```

Fig. 1: An example of semantic clone (Type-4).

In recent years, many neural network-based methods are proposed for detecting such semantic clones, and they have achieved impressive results (both precision and recall are higher than 90%). Most of these neural clone detectors share the same model paradigm:

$$match(c, c') = F(\Phi(c), \Phi(c')) \tag{1}$$

where c and c' are two code fragments, Φ is a learnable neural network that encodes each code fragment as a vector, and F is a function that measures the semantic similarity between two vectors. The code fragment pair (c, c') will be regarded as a clone if and only if $match(c, c')$ is larger than a threshold δ.

Researchers usually define F based on cosine similarity, Euclidean distance, or linear classification. The key in these neural clone detectors is the source code representation method Φ. In CDLH [12], Φ is an AST-based LSTM network. In ASTNN [15], Φ is an AST-based RNN network, in which each large AST is split into a sequence of small statement trees, thus alleviating the long-term dependency problem. In TBCCD [16], Φ is an AST-based convolution network. In DeepSim [13], Φ is based on a matrix-based representation which encodes code control flow and data flow.

B. Rethinking Semantic Clone Benchmarks

To evaluate the effectiveness of neural clone detectors, some benchmarks that contain a large number of semantic clones have been proposed and widely used.

As it is challenging for annotators to find in-the-wild semantic clones from large-scale code repositories, these benchmarks were usually created based on specific functionalities. BigCloneBench [22] is a clone detection benchmark containing 7,868,560 ground truth clones (98.23% of them are semantic clones), while all of them are clones of 43 specific functionalities. To create this benchmark, the researchers began by selecting 43 commonly needed functionalities in open-source Java projects as *target functionalities* (e.g., *Bubble Sort*, *Web Download*, and *Decompress Zip*). Code fragments (i.e., functions) that might implement a target functionality were identified using keywords and source code pattern heuristics, then these identified code fragments were manually tagged as true or false positive of the target functionality by judges. All true positive code fragments of a functionality form a large clone group. OJClone [23] is a dataset that contains 104 functionalities. Specifically, each functionality is a programming question on OpenJudge[2], and there are 500 corresponding solutions (submitted by students, written in C, passing all test cases) for each functionality. Originally this dataset was created for program classification, but researchers also widely used it as a clone detection benchmark: two code fragments (i.e., solutions) are regarded as a ground truth clone if and only if they are solutions of the same functionality. GCJ [13] is a benchmark similar to OJClone. It contains 1,669 solutions (written in Java) for 12 different functionalities (i.e., programming questions from Google Code Jam contests[3]).

We carefully rethink the impact of *specific functionalities* on the evaluation of neural clone detection. Our main concern is that: neural networks may just learn how to classify these specific functionalities, rather than how to detect semantic clones. This concern origins from the fact that neural clone detectors rely on a large number of true/false clones for training. The standard evaluation methodology is to divide data into disjoint training and test sets. However, the diversity of functionalities is limited in these benchmarks, leading to the result that: for each code fragment in test set, there are always many code fragments in training set that have the same functionality as it. Therefore, though neural clone detectors achieved good performance in this experimental setup, a possible reason for the good performance is that neural networks learn to represent these specific functionalities well.

[2]http://openjudge.cn/
[3]https://codingcompetitions.withgoogle.com/codejam

This is not a true evaluation of neural clone detection, as it is an essential need that clone detectors should find clones of various functionalities, rather than just specific functionalities that have been previously observed in training set. The current evaluation methodology cannot tell us about a neural clone detector's generalizability to handle code fragments of unseen functionalities. Therefore, it is necessary to improve the evaluation methodology and revisit existing neural clone detectors based on it.

III. Evaluating Generalizability

A. An Improved Evaluation Methodology

To evaluate the cross-functionality generalizability of neural clone detection, we propose an improved evaluation methodology. The intuition is simple yet effective: training/test sets in current benchmarks should be re-partitioned, with the restriction that no functionality is allowed to appear in both of them.

Here we give the formalism description. D is a dataset that contains many code fragments: $C = \{c_1, c_2, ..., c_{|C|}\}$. F is the set of functionalities: $F = \{f_1, f_2, ..., f_{|F|}\}$. Take OJClone as an example: we have $|F| = 104$ and $|C| = 104 \times 500$. $L : C \rightarrow F$ is a function that indicates the functionality of each code fragment. For each dataset, L is a known function that is determined from the collection procedure of the dataset.

To obtain training/test sets for evaluating neural clone detectors, we propose the following 3 steps:

1) *Creating Functionality Groups*. We divide functionalities (F) into K disjoint groups as evenly as possible (K is a hyper-parameter). We denote these groups as G_1, G_2, ..., G_K.
2) *Creating Training-Test Grid*. For each $1 \leq i, j \leq K$, we set an experiment $E_{i,j}$ in which neural clone detectors are trained on functionalities in G_i and tested on functionalities in G_j. Therefore, we have $K \times K$ experiments with different training-test functionality groups, and we form them as a grid. We define that an experiment is an *unseen-functionality experiment*, if $G_i \neq G_j$; otherwise we define this experiment as a *seen-functionality experiment*.
3) *Sampling Code Fragment Pairs*. For each functionality group $G_i(1 \leq i \leq K)$, we randomly sample 3 disjoint sets of code fragment pairs from $\{(c_x, c_y, I(c, c'))|c, c' \in C; L(c), L(c') \in G_i; c \neq c'\}$, where I is an indicator function:

$$I(c, c') = \begin{cases} 0 & L(c) \neq L(c') \\ 1 & L(c) = L(c') \end{cases} \tag{2}$$

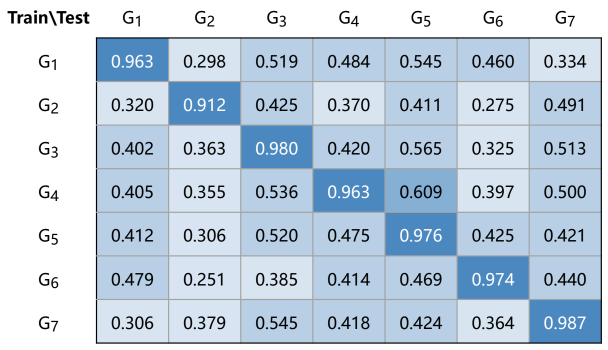
We denote these 3 sets as P_i^{train}, P_i^{dev}, and P_i^{test}, respectively. For each experiment $E_{i,j}(1 \leq i, j \leq K)$, neural clone detectors will be trained on P_i^{train}, validated on P_j^{dev}, and then tested on P_j^{test}.

Neural clone detectors formulate semantic clone detection as a binary-classification task and use precision/recall/F1-score as evaluation metrics. Previous studies proved that existing

TABLE I: Performance of two neural clone detectors when generalizing to unseen functionalities

	Average F1 Score							
	G_1	G_2	G_3	G_4	G_5	G_6	G_7	AVG
ASTNN								
Seen	0.96	0.91	0.98	0.96	0.98	0.97	0.98	0.96
Unseen	0.39	0.33	0.49	0.43	0.50	0.37	0.45	0.42
TBCCD								
Seen	0.98	0.92	0.97	0.97	0.98	0.95	0.98	0.96
Unseen	0.42	0.40	0.56	0.53	0.53	0.51	0.43	0.48

⋆ AFR-rate(ASTNN) = 0.44, AFR-rate(TBCCD) = 0.50



Train\Test	G1	G2	G3	G4	G5	G6	G7
G1	0.963	0.298	0.519	0.484	0.545	0.460	0.334
G2	0.320	0.912	0.425	0.370	0.411	0.275	0.491
G3	0.402	0.363	0.980	0.420	0.565	0.325	0.513
G4	0.405	0.355	0.536	0.963	0.609	0.397	0.500
G5	0.412	0.306	0.520	0.475	0.976	0.425	0.421
G6	0.479	0.251	0.385	0.414	0.469	0.974	0.440
G7	0.306	0.379	0.545	0.418	0.424	0.364	0.987

Fig. 2: F1 results of ASTNN trained on different functionality groups (y-axis) and tested on different functionality groups (x-axis). Each cell is colored according to F1 score: the deeper a cell is colored, the better the neural clone detector performs in the corresponding experiment setting ($E_{i,j}$).

neural clone detectors can achieve very high performance on $E_{i,j}$ if $i = j$. However, we cannot conclude that these neural code detectors well learn how to find semantic clones, or they just learn how to classify specific functionalities in G_i. Therefore, we need to report P/R/F1 results for each $E_{i,j}(i \neq j)$. As this may involve many result numbers, we further introduce **Average F1 Remaining Rate** (AFR rate), a new metric to summarily evaluate the cross-functionality generalizability of neural clone detection:

$$\text{AFR rate} = \frac{\sum_{i,j}^{i \neq j} F1(E_{i,j})}{(K-1) \cdot \sum_i F1(E_{i,i})} \tag{3}$$

The more AFR rate is higher than 0, the more it indicates that the neural clone detector has cross-functionality generalizability.

B. Experimental Setup

We conduct empirical experiments to evaluate the cross-functionality generalizability of neural clone detection. We choose two state-of-the-art neural clone detectors, ASTNN and TBCCD, as our evaluation objects.

We build our benchmark based on OJClone dataset. This dataset has 104 functionalities, which is much more than BigCloneBench (43 functionalities) and GCJ (12 functionalities). All these 104 OJClone functionalities are divided into 7 groups (G_1, G_2, G_3, ... G_7): G_1 contains functionality

TABLE II: How does training data diversity impact generalizability

	Average F1 Score								AFR rate
	G_1	G_2	G_3	G_4	G_5	G_6	G_7	Average	(w.r.t. baseline)
ASTNN									
training data diversity = 15	0.32	0.30	0.52	0.48	0.55	0.46	0.43	0.44	0
training data diversity = 30	0.50	0.35	0.60	0.61	0.57	0.51	0.56	0.53	+0.09
training data diversity = 45	0.62	0.45	0.68	0.63	0.61	0.51	0.57	0.58	+0.15
training data diversity = 60	0.61	0.45	**0.72**	0.64	0.58	0.61	**0.63**	0.60	+0.17
training data diversity = 75	0.64	0.44	0.71	0.67	**0.71**	**0.62**	0.59	**0.62**	+0.19
training data diversity = 89	**0.66**	**0.46**	0.71	**0.70**	0.66	0.58	0.56	0.61	+0.18
TBCCD									
training data diversity = 15	0.42	0.40	0.56	0.53	0.53	0.51	0.43	0.48	0
training data diversity = 30	0.51	0.47	0.61	0.63	0.64	0.53	0.58	0.57	+0.09
training data diversity = 45	0.61	0.49	0.63	0.63	0.64	0.60	0.63	0.60	+0.13
training data diversity = 60	**0.64**	**0.53**	0.67	0.58	0.59	0.54	**0.65**	0.60	+0.13
training data diversity = 75	0.62	0.49	0.65	0.65	0.63	**0.66**	**0.65**	0.62	+0.15
training data diversity = 89	0.63	0.52	**0.68**	**0.69**	**0.66**	0.65	0.61	**0.63**	+0.16

IDs 1-15, G_2 contains functionality IDs 16-30, and so on. G_7 only contains 14 functionalities (IDs 91-104). For each group $G_i (1 \leq i \leq 7)$, we sample 30,000/10,000/10,000 code fragment pairs as P_i^{train}/P_i^{dev}/P_i^{test}.

C. Results and Observations

Table I shows the performance of two neural clone detectors when generalizing to unseen functionalities. Columns $G_1, G_2, ..., G_7$ represents different functionality groups for test. We use *Seen* to denote that training data are collected from the same functionality group as the test set, and we use *Unseen* to denote that the training set share no common functionality with the test set. For example, for ASTNN, we have $\sum_{i=2}^{7} F1(E_{i,1})/6 = 0.39$. Figure 2 shows detailed F1 results of ASTNN trained on different functionality groups (y-axis) and tested on different functionality groups (x-axis).

Our observations are as follows:

- **Good ability for modeling seen functionalities**. For each experiment in which all test functionalities have been previously observed in training set (i.e., 7 experiments on the diagonal from top left to bottom right in Figure 2), ASTNN achieves very high performance. All the 7 experiments have F1 score higher than 0.9, and the average F1 score is 0.96. These results are consistent with the results reported in the original ASTNN paper. However, as discussed in Section II-B, we argue that results on such experiment settings can only indicate that the neural clone detector's ability to represent code fragments of seen functionalities, but not the true ability to detect semantic clones that may involve unseen functionalities.
- **Cannot well generalize to unseen functionalities**. From experiments outside the aforementioned diagonal (i.e., $E_{i,j}$ for each $1 \leq i, j \leq 7$ and $i \neq j$), we can observe that ASTNN cannot generalize to unseen functionalities as expected. F1 scores of these experiments range from 0.251 ($E_{6,2}$) to 0.609 ($E_{4,5}$). The average F1 score is 0.423. These results indicate that: the essence of the learned models is likely to be program classification, rather than clone detection. Therefore, it is difficult to use ASTNN in real-world code clone detection scenarios.

In TBCCD, our observations are the same as those in ASTNN.

Finding: To evaluate neural clone detectors, we need to **minimize the functionality overlap between training set and test set**, thus truly indicating the generalizability for detecting real-world semantic clones.

The ideal way to minimize functionality overlap is to collect semantic clones in the wild, rather than specifying several target functionalities in advance (just as BigCloneBench, OJClone and GCJ do). However, this would be too costly for human annotation. Therefore, to evaluate neural clone detectors more efficiently, we make a trade-off: we still need to specify several target functionalities in advance, but the total number of target functionalities should be as many as possible (e.g., $\geq$ 100), and the training/test set should be split based on functionalities. Previous researches using random training/test splits suffer from serious "functionality leak" problem, resulting in models achieving almost perfect evaluation results exhibit poor real-world performance. Our proposed evaluation methodology addresses this problem, thus can better indicating the real performance of neural clone detectors (though it is still not as solid as evaluating in the wild).

IV. Key Factors of Generalizability

In this section, we explore to understand key factors that impact the cross-functionality generalizability of neural clone detection. Specifically, we mainly investigate 3 potential directions: (1) training data diversity, (2) vocabulary, and (3) locality.

A. Training Data Diversity

To investigate key factors of cross-functionality generalizability, one hypothesis is that:

H1. *The cross-functionality generalizability of neural clone detection can be improved through increasing training data diversity.*

Here we define *training data diversity* as the total number of functionalities in training set.

This hypothesis is proposed based on the fact that each of our experiments in Section III-B uses only one functionality group for training, i.e., training data diversity is 14 (G_7) or 15 (G_1-G_6). A possible reason for the lack of cross-functionality generalizability is that: neural clone detectors are likely to degenerate to program classifiers for specific functionalities when they are trained on a dataset with small functionality diversity; this problem may be alleviated or addressed through increasing training data diversity.

It is essential to study this hypothesis: if cross-functionality generalizability can be significantly improved through increasing training data diversity, an important direction for future work is to improve the data collection methodology for better functionality diversity; otherwise, it indicates that we cannot equip existing neural clone detectors with true cross-functionality generalizability through collecting much more training data, thus future work should focus on improving these neural model architectures.

We conduct a series of experiments to study this hypothesis. These experiments are set up based on the following steps:

1) Select a functionality group for test. Here we use G_1 as an example.
2) Use G_2 for training, that is, train a neural clone detector (ASTNN/TBCCD) on P^2_{train} and test it on P^1_{test}. In this experiment, the training data diversity is 15.
3) Use $G_2 \cup G_3$ for training. Training data are sampled from pairs of which code fragments are of functionalities in $G_2 \cup G_3$, i.e., the training data diversity is 30. We use the same sampling amount as P^2_{train} (i.e., 30,000). We also keep the same positive rate in training data (i.e., $1/15$) to prevent suffer from the class imbalance problem caused by the growth of training data diversity.
4) Use $G_2 \cup G_3 \cup G_4$ for training. The training data diversity is 45.
5) Use $G_2 \cup G_3 \cup G_4 \cup G_5$ for training. The training data diversity is 60.
...
6) Draw results of the above 6 experiments as a line chart of the influence of training data diversity on clone detection performance (F1 score).

We use $G_1, G_2, ..., G_7$ for test respectively, thus we draw 7 lines in the line chart. Figure 3 shows the results of ASTNN. Our observations are as follows:

- *Increasing training data diversity can significantly improve the cross-functionality generalizability of neural clone detection*. For example, consider ASTNN G_1: the F1 score is 0.320 when the training data diversity is 15; the F1 score will increase to 0.661 when the training data diversity is 89. An increase of 0.341 is observed. For $G_1, G_2, ..., G_7$, the increase ranges from 0.115 to 0.341, and the average increase is 0.183.
- *There are significant marginal effects to improve neural clone detection through increasing training data diversity*. For example, consider ASTNN G_1: the F1 score increases from 0.320 to 0.617 when the training data diversity increases from 15 to 45, while the increase is only 0.044 (from 0.617 to 0.661) when the training data diversity increases from 45 to 89. For $G_1, G_2, ..., G_7$, the average F1 increase for training data diversity $15 \rightarrow 45$ is 0.146, accounting for 79.8% of the increase for $15 \rightarrow 89$. We regard these results as significant marginal effects, which indicates that it is not likely to be sustainable that improving neural clone detection through continuously increasing training data diversity.

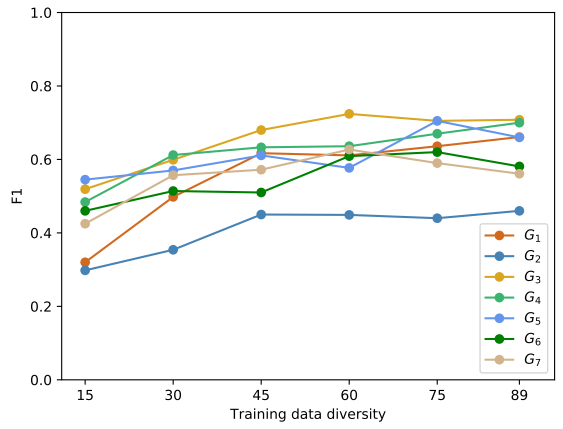


Fig. 3: Influence of training data diversity on performance of neural clone detection (ASTNN). We can observe that: increasing training data diversity can significantly improve the performance of neural clone detection, but there are significant marginal effects.

Table II lists detailed results of both ASTNN and TBCCD. In TBCCD, our observations are the same as those in ASTNN.

Finding: To train neural clone detectors, **training set with diverse functionalities** can alleviate the problem of lacking cross-functionality generalizability. However, due to the marginal effect, this is not a silver bullet to completely address this problem.

B. Vocabulary

Common wisdom suggests that vocabulary is a key factor that may impact the effectiveness of neural networks, especially for source code modeling ([16], [24], [25]). Neural clone detectors need to represent tokens as numerical representations so that the lexical information can be fed into the neural networks. These tokens include reserved words in programming languages (e.g., "*if*", "*int*", and "*break*"), built-in functions and data structures (e.g., "*abs*", "+", and "*vector*"), programmer-defined identifiers (e.g., "*x*", "*y*", and "*max_distance*"), etc. In general methods, a static vocabulary is extracted from training set (mainly based on token frequency), then all tokens which are not in this vocabulary will be converted to a specific token: "<*unknown*>". This works well in natural language processing, but may be problematic for source code. This is mainly because that programmers are free to create various

TABLE III: How does vocabulary impact generalizability

	Average F1 Score								AFR rate
	G_1	G_2	G_3	G_4	G_5	G_6	G_7	Average	(w.r.t. baseline)
ASTNN									
test data average UNK rate = 13.7%	0.39	0.33	0.49	0.43	0.50	0.37	0.45	0.42	0
test data average UNK rate = 11.5%	0.41	0.34	0.51	0.47	0.51	0.37	0.44	0.44	+0.02
test data average UNK rate = 10.5%	0.42	0.36	0.58	0.52	0.52	0.38	0.45	0.46	+0.04
test data average UNK rate = 9.4%	**0.46**	**0.39**	**0.64**	**0.58**	**0.55**	**0.42**	**0.50**	**0.51**	+0.09
TBCCD									
with PACE	0.42	0.40	0.56	0.53	0.53	0.51	0.43	0.48	0
w/o PACE	0.26	0.25	0.36	0.31	0.36	0.34	0.31	0.31	-0.18

tokens (especially variable names and function names), thus aggravating the *out-of-vocabulary* (OOV) problem. That is, in test set, a large amount of tokens will be converted to <*unknown*>, thus the lexical information they carry will be lost. Therefore, an intuitive hypothesis is that:

H2. *the cross-functionality generalizability of neural clone detection can be improved through addressing the OOV problem caused by vocabulary.*

In TBCCD, *Position-Aware Character Embedding (PACE)*, a simple yet effective method for alleviating the OOV problem in source code, is proposed. Therefore, TBCCD does not suffers from the OOV problem. The key point of PACE is to not treat each token as an individual building block, but a position-weighted combination of characters one-hot embeddings. That means for a token that has k characters denoted as $c_1, c_2, ..., c_k$, its embeddings can be obtained with equation $\sum_{i=1}^{k} \frac{k-i+1}{k} \times emb\left[c_i\right]$, where $emb[c_i]$ is the one-hot embedding of c_i. To summarize, PACE learns character embeddings, then generates the embedding of each word by assembling embeddings of every characters in this word. Therefore, PACE addresses the OOV problem, at the cost of lower capability of word-level semantics. The TBCCD paper reported that: though the effectiveness of PACE is marginal in random training/test splits, it can bring significant gain in cross-functionality splits.

We conduct an ablation experiment in which PACE is replaced by a regular token embedding layer (i.e., *TBCCD w/o PACE* in Table III) and the result shows that the cross-functionality generalizability is significantly reduced without PACE. This indicates that the OOV problem brought by vocabulary is likely to be a key factor of generalizability. ASTNN uses a regular token embedding layer (the vocabulary is defined as top 3,000 frequent tokens in training set, and token embeddings are pre-trained using *word2vec*), thus it may suffer from the OOV problem. When we apply PACE to ASTNN, we observe no improvement. This indicates that PACE is not a universal solution to the OOV problem in all model architectures.

We speculate that the reason is: ASTNN requires a larger capability of word-level semantics than TBCCD. TBCCD is a tree-based CNN model, which mainly captures program semantics from AST structures (words are also important, yet secondary). Therefore, for TBCCD, addressing the OOV problem at the cost of lower capability of word-level semantics will do more good than harm. Unlike TBCCD, ASTNN is an RNN-based model, in which each code fragment are pre-processed as a sequence, rather than a tree. Therefore, word-level semantics plays a more important role in ASTNN than in TBCCD. Though PACE can address the OOV problem, this benefit is offset by its lower capability of word-level semantics.

We investigate the impact of vocabulary in ASTNN through breaking down test sets according to the percentage of <*unknown*> tokens. Specifically, we create test sets that contain less <*unknown*> tokens than $P_i^{test} (1 \leq i \leq 7)$. Our assumption is: if ASTNN performs better in test sets with less <*unknown*> tokens, it means that ASTNN can benefit from reducing <*unknown*> tokens, thereby indicating that the OOV problem is a key factor that can impact cross-functionality generalizability.

For each experiment $E_{i,j}$, we re-sample the test set P_j^{test} as follows. For each functionality $f \in G_j$, we sort all code fragments of f by the percentage of <*unknown*> tokens in ascending order. We keep the top 80%/60%/40% of these code fragments. Then, test sets are created from these code fragments. In $P_i^{test} (1 \leq i \leq 7)$, the average UNK rate is 13.7%; In test sets created from 80%/60%/40% code fragments, the average UNK rate is 11.5%/10.5%/9.4%. Therefore, we denote these test settings as "*ASTNN, tested data average UNK rate = 13.7%/11.5%/10.5%/9.4%*" in Table III. We observe that ASTNN has better performance in test cases which less suffer from the OOV problem: on average, the F1 score of "*ASTNN, tested data average UNK rate = 9.4%*" is 0.51, which is much better than the baseline (0.42). This indicates that the problem of lacking cross-functionality generalizability is likely to be partly alleviated by addressing the OOV problem.

Finding: The **out-of-vocabulary problem** is an important factor that limits the cross-functionality generalizability of neural clone detection. Character-level or subword-level token embeddings (e.g., PACE) can help alleviate this problem, but are not universal enough for various neural clone detectors.

C. Locality

We consider a consensus in machine learning community: *local structure inference between two objects is essential for determining the overall inference between these two objects* ([26]–[29]). For example, in natural language processing, if we

TABLE IV: How does locality impact generalizability

	Average F1 Score							
	G_1	G_2	G_3	G_4	G_5	G_6	G_7	AVG
ASTNN	**0.39**	0.33	**0.49**	**0.43**	**0.50**	**0.37**	0.45	**0.42**
with locality	**0.39**	**0.36**	0.47	0.41	0.43	0.33	**0.50**	0.41

want to learn a neural network model to determine whether two natural language sentences are semantically equivalent, this model needs to employ some forms of alignment to associate the relevant local structures (e.g., words, phrases, and clauses) between two sentences.

In code clone detection task, the objects are code fragments, and the local structures can be statements, code blocks, sub-ASTs, etc. Figure 4 shows an example of local structure alignment between code fragments. Intuitively, suppose that code fragment A is semantically equivalent to code fragment B, it is likely that some local structures in A can be aligned with some local structures in B. Local structure alignment is more in line with human perception of code clone detection, thus preventing neural clone detectors degenerate to program classifiers.

Based on this intuition, we hypothesize that:

H3. *incorporating locality into model architecture can help improve the cross-functionality generalizability of neural clone detection.*

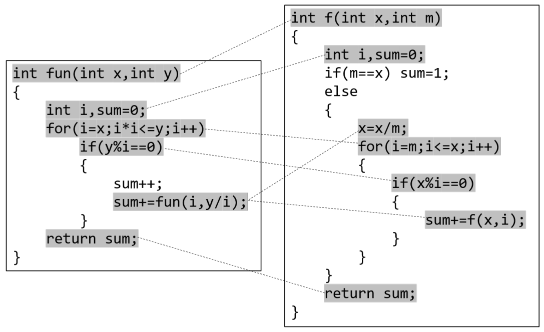
Some recent research works of neural clone detection have incorporate locality into their model architectures ([18], [19]), but it is not easy to adapt their codes to OJClone. Therefore, to investigate this hypothesis, we use ASTNN as the base model architecture and add a locality component on top of it based on common practices in machine learning community.

We briefly summarize the workflow of ASTNN as follows:

1) A code fragment will be parsed into an AST, then the AST will be split into a sequence of statement trees (ST-trees, which are trees consisting of statement nodes as roots and corresponding AST nodes of the statements).
2) For each code fragment, all ST-trees in it are encoded into individual vectors, denoted as $\boldsymbol{e}_1, ..., \boldsymbol{e}_t$. Then, ASTNN uses a Bidirectional Gated Recurrent Unit (Bi-GRU) network to obtain their contextual vectors, denoted as $\boldsymbol{h}_1, ..., \boldsymbol{h}_t$.
3) Code fragment representation is computed by max pooling of $\boldsymbol{h}_1, ..., \boldsymbol{h}_t$, then whether a pair of code fragments is a clone is determined by Euclidean distance between their representations.

We incorporate locality into ASTNN through introducing attention-based alignment between contextual vectors of ST-trees ($\boldsymbol{h}$). Notice that: we do not propose a novel locality component for improving neural clone detection; instead, the goal of this part is to leverage a state-of-the-art locality component [27] (of which the effectiveness has been well proved in machine learning community) to study whether locality is a key factor for cross-functionality generalizability.

Suppose that we have two code fragments $c = \{\boldsymbol{h}_1, ..., \boldsymbol{h}_t\}$ and $c' = \{\boldsymbol{h'}_1, ..., \boldsymbol{h'}_{t'}\}$, we add a soft alignment layer to



```
int fun(int x,int y)
{
    int i,sum=0;
    for(i=x;i*i<=y;i++)
        if(y%i==0)
        {
            sum++;
            sum+=fun(i,y/i);
        }
    return sum;
}
```

```
int f(int x,int m)
{
    int i,sum=0;
    if(m==x) sum=1;
    else
    {
        x=x/m;
        for(i=m;i<=x;i++)
        {
            if(x%i==0)
            {
                sum+=f(x,i);
            }
        }
    }
    return sum;
}
```

Fig. 4: An example of local structure alignment between code fragments. Intuitively, suppose that code fragment A is semantically equivalent to code fragment B, it is likely that some local structures (e.g., statements or blocks) in A can be aligned with some local structures in B.

ASTNN:

$$\widetilde{\boldsymbol{h}}_i = \sum_{j=1}^{t'} \frac{exp(\boldsymbol{h}_i \cdot \boldsymbol{h'}_j)}{\sum_{k=1}^{t'} exp(\boldsymbol{h}_i \cdot \boldsymbol{h'}_k)} \boldsymbol{h'}_j, \forall i \in [1, ..., t] \tag{4}$$

Intuitively, in Equation 4, the content in $\{\boldsymbol{h}'_j\}_{j=1}^{t'}$ that is relevant to $\boldsymbol{h}_i$ will be selected and represented as $\widetilde{\boldsymbol{h}}_i$. The same is performed for each ST-tree in c'.

Then, we use a Bi-GRU network to convert local alignment information ($\widetilde{\boldsymbol{h}}$) to local alignment-aware contextual representations ($\hat{\boldsymbol{h}}$):

$$\boldsymbol{m}_i = [\boldsymbol{h}_i; \widetilde{\boldsymbol{h}}_i; \boldsymbol{h}_i - \widetilde{\boldsymbol{h}}_i; \boldsymbol{h}_i \otimes \widetilde{\boldsymbol{h}}_i] \tag{5}$$

$$\hat{\boldsymbol{h}}_1, ..., \hat{\boldsymbol{h}}_t = \text{Bi-GRU}(\boldsymbol{m}_1, ..., \boldsymbol{m}_t) \tag{6}$$

Representation of code fragment c will be computed by max pooling of $\hat{\boldsymbol{h}}_1, ..., \hat{\boldsymbol{h}}_t$, and the remaining steps remain exactly the same as ASTNN.

Table IV shows the experimental results. We disappointedly find that the cross-functionality generalizability of neural clone detection cannot benefit from incorporating with the locality component. We speculate that the reasons may be three-folds: (1) Code fragments that are semantically equivalent may have totally different syntactic structures (as Figure 1 shows), making neural networks difficult to leverage local structure alignment information. (2) The OOV problem discussed in Section III also lead to lots of noisy alignments. (3) The locality component we use is more suitable for sequence data (i.e., natural language sentences) rather than tree data (i.e., ASTs of code fragments), thus tree-based locality component may be a potential direction for improvement.

Finding: In neural clone detection, local structure alignment has NOT yet been well leveraged to improve cross-functionality generalization.

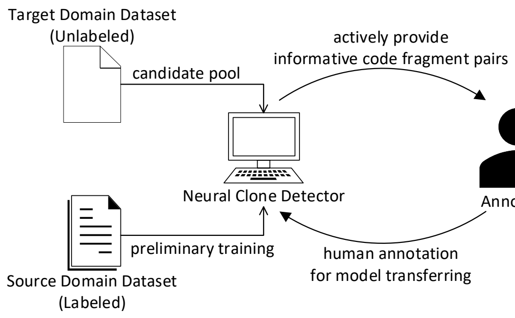


Fig. 5: Human-in-the-loop mechanism for domain adaptation of neural clone detection.

Algorithm 1: Human-in-the-Loop Mechanism

```
Input: $D_{src}$ (annotated dataset of source domain),
       $C_{trg}$ (code fragments of target domain),
       $Oracle$ (human annotators), $M$ (active query
       batch size), $V$ (informative degree
       measurement function)
Output: $model$ (a neural clone detector for target
        domain)
1 $model \leftarrow TrainModel(D_{src})$
2 $pool \leftarrow$ candidate code fragment pairs from $C_{trg}$
3 $D_{trg} \leftarrow \emptyset$
4 while budget not exhausted do
5     $queries \leftarrow$ top $M$ pairs in $pool$ by $V(model, pair)$
6     $D_{trg} \leftarrow D_{trg} \cup Oracle(queries)$
7     $model \leftarrow TrainModel(D_{src} \cup D_{trg})$
8 end
```

V. Human-in-the-Loop for Domain Adaptation

A. Task: Domain Adaptation

From Section III and IV we can conclude that: neural clone detectors cannot well generalize to unseen functionalities as expected. Though we find that this problem can be alleviated through addressing the OOV problem caused by vocabulary and increasing training data diversity, there is still a large performance gap (AFR rate is about 46%/50% for ASTNN/T-BCCD). Therefore, it is not a good idea to train existing neural clone detectors on a common dataset and directly use them to find semantic clones in various code repositories that contain lots of functionalities that have never been previously observed in the training set. Instead, for each individual code repository, if we want to find semantic clones effectively (with high precision and recall), we have to annotate a specific training set for this code repository. This is expensive, thus limiting the scalability of neural clone detection in real-world scenarios.

We formulate this as a domain adaptation problem, that is, how to adapt a model (i.e., neural clone detector) learned from one domain (i.e., the training set) to a new domain (i.e., a new code repository containing lots of unseen functionalities). Here we define that two domains are different if they contain different functionalities.

B. Solution: Human-in-the-Loop

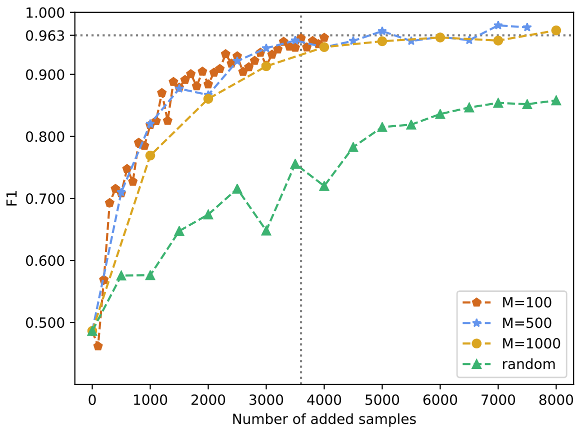
We propose a human-in-the-loop mechanism (see Figure 5) to address this domain adaptation problem. The key point is *"a little annotation does a lot of good"*, based on the combination of transfer learning and active learning. Given a neural clone detector learned from a high-resource domain, our goal is to adapt it to a unlabeled target domain. To achieve this, our human-in-the-loop mechanism automatically explores the target domain and actively select most informative (rather than random) code fragment pairs that can help transfer the neural clone detector learned from source domain to the target domain. Human annotate these actively selected code fragment pairs (whether this pair is a clone or not), then these annotated data are leveraged as new training samples to update the neural clone detector. This mechanism helps domain adaptation of neural clone detection through improving annotation efficiency: human just need to annotate a small collection of samples (which is much smaller than regular annotation) to achieve an accurate neural clone detector for the low-resource target domain.

Algorithm 1 is an overall procedure of this human-in-the-loop mechanism. It exploits both transfer learning and active learning to improve annotation efficiency. In the following we explain this algorithm from these two aspects.

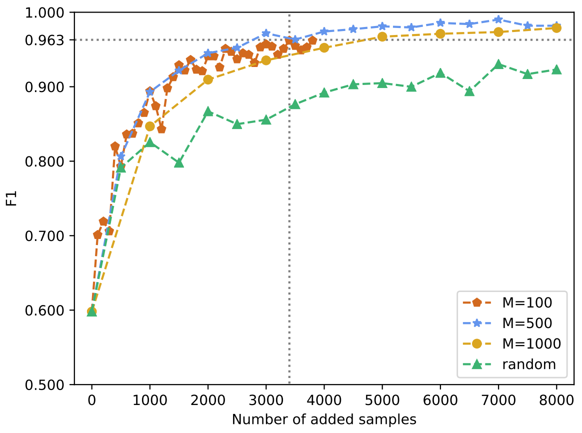
1) Transfer Learning: The goal of transfer learning is to leverage knowledge learned from source domain and apply it to target domain. In Algorithm 1, we train an neural clone detector on source domain as our preliminary model (Line 1). Once the oracle (i.e., human annotators) provides some annotated code fragment pairs in target domain, these data will be used to update the model (Line 7). Therefore, the neural clone detector will be transferred from source domain to target domain iteratively. There are two optional strategies for model update: one is to iteratively fine-tune the model with newly annotated data, and the other is to learn from scratch (i.e., re-train the model on $D_{src} \cup D_{trg}$) in each iteration. In the fine-tuning strategy, there are some hyper-parameters such as learning rate and the number of fine-tuning epochs, and it is not easy to find the best setting of these hyper-parameters for each scenario (in different scenarios, the best hypher-parameter settings are usually different). Our preliminary experiments show that results of these two model update strategies are comparable, thus we finally choose the learning-from-scratch strategy (Line 7) as it is much simpler than fine-tuning.

2) Active Learning: After training a preliminary model from source domain, we start the active learning process based on this model's outputs.

The first step of active learning is to create a pool consisting of many candidate (unlabeled) code fragment pairs from target domain (Line 2). There are 3 optional strategies for creating this pool: it can be all possible code fragment pairs in the target domain, if the total number of code fragments in this domain is not large; or we can just use a part of it through simple



(a) Performance when the source domain is G_1 and the target domain is G_3



(b) Performance when the target domain is G_3, and the source domain is $G_1 \cup G_2 \cup G_4 \cup G_5 \cup G_6 \cup G_7$

Fig. 6: Human-in-the-loop mechanism performance. The horizontal dashed line is the performance of the model trained on 30,000 annotated samples from target domain. Reaching the horizontal dashed line means that our human-in-the-loop mechanism successfully transfer the neural clone detector (ASTNN) to the target domain.

random-sampling; in real-world scenarios, considering that clones are sparse in randomly sampled code fragment pairs, we can also use some heuristics to filter out code fragment pairs that are unlikely to be clones before sampling (e.g., whether two code fragments have the same input/output data types).

After that, we actively select M code fragment pairs from the pool (Line 5). Human annotators label whether each of these M pairs is a clone or not (Line 6), then these newly annotated data are used for transfer learning (Line 7). This procedure iterates until the budget for data annotation (usually very limited) is exhausted.

The key in active learning is the informative degree measurement function V. The actively selected pairs should be highly informative, thus they can help the model be transferred to target domain efficiently. In this paper, we compute $V(model, pair)$ based on model uncertainty [56]:

$$V(model, pair) = |1 - \max_{l \in \{+,-\}} P(l|model, pair)| \tag{7}$$

$P(+|model, pair)$ is the model's estimated probability that this pair is a clone, and we have $P(-|model, pair) = 1 - P(+|model, pair)$. Intuitively, $V(model, pair)$ can be regarded as the uncertainty of the model given the pair. The higher the uncertainty is, the more it means that this pair is likely to help improve the model. Therefore, in each active learning iteration, we select candidate code fragment pairs with top M uncertainty for human annotation.

Besides these uncertainty-based methods, many methods have been proposed in the literature to better find informative samples for active learning. For example, some methods are based on representative [57], and some others are based on diversity [58]. In this paper, we think our uncertainty-based method is simple and effective enough, thus we leave the exploration of other sample selecting methods to future work.

C. *Simulation Experiments*

We conduct simulation experiments to evaluate the effectiveness of our human-in-the-loop mechanism.

1) Setup: We use ASTNN as the model architecture. Evaluation benchmark is created based on OJClone. As discussed in Section III-A, we have divided all the 104 functionalities in OJClone into 7 groups and denote them as $G_1, G_2, ..., G_7$. For each group $G_i (1 \leq i \leq 7)$, we have created trainning/development/test sets ($P_i^{train}/P_i^{dev}/P_i^{test}$) for it. We select one functionality group G_{src} as source domain and another functionality group G_{trg} as target domain for simulation experiment. Following Algorithm 1, the ASTNN model will be initially trained on P_{src}^{train}, then iteratively updated by samples actively selected from P_{trg}^{train} (each iteration has M samples), and finally validated/tested on $P_{trg}^{dev}/P_{trg}^{test}$. To investigate the impact of the hyper-parameter M (i.e., active query batch size), we evaluate the human-in-the-loop mechanism with different M values: 100/500/1,000.

2) Results and Analysis: Figure 6 shows the performance of our human-in-the-loop mechanism in one simulation experiment. In Figure 6a, we randomly select G_1 as the source domain and G_3 as the target domain. The "*random*" curve means that the samples for annotation (i.e., *queries* in Algorithm 1) are randomly sampled from *pool*, rather than actively selected by *model*. It is used as an ablation study for proving the effectiveness of active learning. The horizontal dashed line is the performance of the model trained on 30,000 annotated samples from target domain. If a curve reaches this horizontal line when the x-axis value is X, it indicates that our human-in-the-loop mechanism successfully transfer the neural clone detector from source domain to target domain with X annotated samples for target domain.

From Figure 6a, we can observe that:

1) Our human-in-the-loop mechanism can significantly reduce data annotation efforts. When $M = 100/500$, it

requires about 3600 annotations to transfer the ASTNN model to the target domain. Comparing to P_{trg}^{train} (which has 30,000 annotated samples in total), we reduce the amount of annotations by about 88%.

2) Selecting informative candidate samples actively is important in this human-in-the-loop mechanism. When samples for annotation are randomly sampled (the *random* curve) rather than actively selected (curves $M = 100/500/1000$), the performance significantly drops.

We also conduct another simulation experiment to examine how this human-in-the-loop mechanism performs on training data with higher diversity. In this experiment, the target domain is G_3, and the source domain is $G_1 \cup G_2 \cup G_4 \cup G_5 \cup G_6 \cup G_7$ (training data diversity = 89). Figure 6b shows the results. In Figure 6b, we have the same observations as in Figure 6a.

Finding: To adapt neural clone detection to real-world low-resourced domains, our **human-in-the loop** mechanism can effectively **reduce annotation efforts**.

Note that this experiment is simulated, which means that we already have these annotations, but only use some of them to simulate the human annotation process. This approach alleviates the problem of lacking cross-functionality generalizability, but it is not a cure for the problem: it can just reduce the annotation cost, but still require users to label data in the target domains.

VI. Threats to Validity

We have identified the following main threats to validity:

- *Programming languages.* Our experiments are conducted based on OJClone dataset, in which code fragments are written in C. There are some other clone detection datasets (e.g., BigCloneBench and GCJ), which are mainly in Java language. The reasons why we only conduct experiments on OJClone is to prevent issues that may caused by the lack of functionality diversity (e.g., GCJ dataset only has 12 functionalities) and data imbalance (e.g., in BigCloneBench, the functionality "*Copy File*" accounts for 54.3% of clone pairs, and the top 10 functionalities accounts for 91.7%). In principle, neural clone detectors are designed based on generalized program structures (e.g., AST and PDG), rather than specific programming languages. Therefore, we can speculate that our findings in this paper can generalize to other programming languages. In the future, we will work to improve clone detection benchmarks in terms of functionality diversity and programming language diversity.
- *Model evaluated.* In this paper, we choose two state-of-the-art neural clone detectors, ASTNN and TBCCD, as our evaluation objects. ASTNN is the representative of RNN-based models, and TBCCD is the representative of CNN-based models. Therefore, we think findings in this paper can generalize to other RNN/CNN-based models, which accounts for a large part of previous work. However, we still need to further explore whether recent methods based on GNNs (Graph Neural Networks) or PLMs (pretrained language models) have better cross-functionality generalizability or not. Though currently we do not know whether our findings got from RNN/CNN-based models can apply to GNN/PLM-based models, we suggest that newly proposed neural clone detectors should be evaluated based on a cross-functionality methodology to alleviate threats to validity.

VII. Conclusion

In this work, we identify a key oversight in the current evaluation methodology for neural clone detection: cross-functionality generalizability (i.e., the ability to detect semantic clones of which the functionalities have never been previously observed in the training dataset). Our contributions are 3-folds: (1) By proposing an evaluation methodology for cross-functionality generalizability, we conduct experiments on two state-of-the-art neural clone detectors, and find that they cannot well generalize to unseen functionalities as expected. (2) To understand key factors that impact the cross-functionality generalizability, we conduct empirical analysis on 3 factors (training data diversity, vocabulary, and locality), and find that the performance loss on unseen functionalities can be reduced through increasing training data diversity and addressing the out-of-vocabulary problem. (3) To adapt neural clone detectors to new code repositories containing lots of unseen functionalities, we propose a human-in-the-loop mechanism that helps reduce the amount of annotations by about 88%.

Our analysis has clear implications for future work: (1) New neural code clone detectors should be evaluated on functionality-based data splits to ensure that they can generalize to real-world scenarios. (2) Based on findings in this paper, future research directions for improving cross-functionality generalizability include: addressing the OOV problem; exploring better model architectures that have the capability to benefit from more diverse training data; exploring unsupervised or semi-supervised methods that can exploit large-scale unlabeled code fragments to improve neural clone detection. (3) Human-in-the-loop is an efficient way to adapt neural clone detection to low-resource real-world code repositories, and a potential research direction is to explore better algorithms for selecting informative code fragment pairs. (4) More broadly, these implications are not limited to neural clone detection, but also various neural source code representation methods.

Acknowledgments

The work was supported by the National Key Research and Development Program of China (No. 2019YFB1704003), the National Nature Science Foundation of China (No. 71690231 and No. 62021002), Tsinghua BNRist and Beijing Key Laboratory of Industrial Bigdata System and Application.

References

[1] P. Weissgerber and S. Diehl, "Identifying refactorings from source-code changes," in *21st IEEE/ACM international conference on automated software engineering (ASE'06)*. IEEE, 2006, pp. 231–240.

[2] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida, "Shinobi: A tool for automatic code clone detection in the ide," in *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 313–314.

[3] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, and T. Xie, "Xiao: tuning code clones at hands of engineers in practice," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012, pp. 369–378.

[4] N. A. Milea, L. Jiang, and S.-C. Khoo, "Vector abstraction and concretization for scalable detection of refactorings," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 86–97.

[5] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 485–495.

[6] E. Juergens, F. Deissenboeck, and B. Hummel, "Clonedetective-a workbench for clone detection research," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 603–606.

[7] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, and J. Streit, "Can clone detection support quality assessments of requirements specifications?" in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, 2010, pp. 79–88.

[8] J. Doe, "Recommended practice for software requirements specifications (ieee)," *IEEE, New York*, 2011.

[9] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of computer programming*, vol. 74, no. 7, pp. 470–495, 2009.

[10] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–9.

[11] M. Rieger, "Effective clone detection without language barriers," Ph.D. dissertation, Verlag nicht ermittelbar, 2005.

[12] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code." in *IJCAI*, 2017, pp. 3034–3040.

[13] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 141–151.

[14] H. Wei and M. Li, "Positive and unlabeled learning for detecting software functional clones with adversarial training." in *IJCAI*, 2018, pp. 2840–2846.

[15] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[16] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang, "Neural detection of semantic code clones via tree-based convolution," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 70–80.

[17] Y.-Y. Zhang and M. Li, "Find me if you can: Deep software clone detection by exploiting the contest between the plagiarist and the detector," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 5813–5820.

[18] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 2019, pp. 3835–3845. [Online]. Available: http://proceedings.mlr.press/v97/li19d.html

[19] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 261–271.

[20] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 131–140.

[21] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 1157–1168.

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

[23] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16. AAAI Press, 2016, p. 1287–1293.

[24] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 763–773.

[25] R. M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big Code != Big Vocabulary: Open-Vocabulary Models for Source code," in *Proceedings of the 42nd International Conference on Software Engineering*, ser. ICSE '20. ACM, 2020. [Online]. Available: https://doi.org/10.1145/3377811.3380342

[26] A. Parikh, O. Täckström, D. Das, and J. Uszkoreit, "A decomposable attention model for natural language inference," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 2249–2255. [Online]. Available: https://www.aclweb.org/anthology/D16-1244

[27] Q. Chen, X. Zhu, Z. Ling, S. Wei, H. Jiang, and D. Inkpen, "Enhanced lstm for natural language inference," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL 2017)*. Vancouver: ACL, July 2017.

[28] C. Ciliberto, F. Bach, and A. Rudi, "Localized structured prediction," in *Advances in Neural Information Processing Systems*, 2019, pp. 7301–7311.

[29] T. Sylvain, L. Petrini, and D. Hjelm, "Locality and compositionality in zero-shot learning," in *International Conference on Learning Representations*, 2019.

[30] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[31] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 419–428.

[32] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 334–345.

[33] V. Raychev, P. Bielik, M. Vechev, and A. Krause, "Learning programs from noisy data," *ACM SIGPLAN Notices*, vol. 51, no. 1, pp. 761–774, 2016.

[34] A. Bhoopchand, T. Rocktäschel, E. Barr, and S. Riedel, "Learning python code suggestion with a sparse pointer network," 2016. [Online]. Available: http://arxiv.org/abs/1611.08307

[35] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 2018, pp. 4159–25.

[36] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 2018 40th International Conference on Software Engineering (ICSE 2018)*. ACM, 2018.

[37] Q. Chen and M. Zhou, "A neural framework for retrieval and summarization of source code," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 826–831.

[38] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[39] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 964–974.

[40] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International conference on machine learning*, 2016, pp. 2091–2100.

[41] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual*

Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2016, pp. 2073–2083.

[42] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 397–407.

[43] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 404–419, 2018.

[44] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *International Conference on Learning Representations*, 2018.

[45] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Software Engineering*, vol. 25, no. 3, pp. 2179–2217, 2020.

[46] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[47] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 795–806.

[48] Y. Wang, L. Du, E. Shi, Y. Hu, S. Han, and D. Zhang, "Cocogum: Contextual code summarization with multi-relational gnn on umls," Microsoft, Tech. Rep. MSR-TR-2020-16, May 2020. [Online]. Available: https://www.microsoft.com/en-us/research/publication/cocogum-contextual-code-summarization-with-multi-relational-gnn-on-umls/

[49] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," in *Advances in neural information processing systems*, 2018, pp. 2547–2557.

[50] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *arXiv preprint arXiv:2006.03511*, 2020.

[51] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 297–308.

[52] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018.

[53] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations*, 2018.

[54] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on software engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[55] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[56] A. Culotta and A. McCallum, "Reducing labeling effort for structured prediction tasks," in *AAAI*, vol. 5, 2005, pp. 746–751.

[57] S. Dasgupta and D. Hsu, "Hierarchical sampling for active learning," in *Proceedings of the 25th international conference on Machine learning*, 2008, pp. 208–215.

[58] A. Kirsch, J. van Amersfoort, and Y. Gal, "Batchbald: Efficient and diverse batch acquisition for deep bayesian active learning," 2019.