CCLearner: A Deep Learning-Based Clone Detection Approach

Liuqing Li, He Feng, Wenjie Zhuang, Na Meng and Barbara Ryder
Department of Computer Science, Virginia Tech
Blacksburg, VA, USA
{liuqing, fenghe, kaito, nm8247}@vt.edu, ryder@cs.vt.edu

***Abstract*—Programmers produce code clones when developing software. By copying and pasting code with or without modification, developers reuse existing code to improve programming productivity. However, code clones present challenges to software maintenance: they may require consistent application of the same or similar bug fixes or program changes to multiple code locations. To simplify the maintenance process, various tools have been proposed to automatically detect clones [1], [2], [3], [4], [5], [6]. Some tools tokenize source code, and then compare the sequence or frequency of tokens to reveal clones [1], [3], [4], [5]. Some other tools detect clones using tree-matching algorithms to compare the Abstract Syntax Trees (ASTs) of source code [2], [6]. In this paper, we present CCLEARNER, *the first solely token-based clone detection approach leveraging deep learning*. CCLEARNER extracts tokens from known method-level code clones and non-clones to train a classifier, and then uses the classifier to detect clones in a given codebase.**

To evaluate CCLEARNER, we reused BigCloneBench [7], an existing large benchmark of real clones. We used part of the benchmark for training and the other part for testing, and observed that CCLEARNER effectively detected clones. With the same data set, we conducted *the first systematic comparison experiment between* CCLEARNER *and three popular clone detection tools*. Compared with the approaches not using deep learning, CCLEARNER achieved competitive clone detection effectiveness with low time cost.

***Keywords*-deep learning; clone detection; empirical**

I. INTRODUCTION

Programmers create and maintain code clones in software development. Previous research shows that code clones are common in software systems. For instance, there was about 19% duplicated code in X Window System (X11) [8], and 15-25% duplicated code in Linux kernel [9]. A recent study on 7,800 open-source projects written in C or C++ shows that 44% of the projects have at least one pair of identical code snippets [10]. By copying and pasting code with or without modification, developers reuse existing code to improve programming productivity. On the other hand, code clones also present challenges to software maintenance: they may require consistent application of the same or similar bug fixes or program changes to multiple code locations.

There are four main types of code clones defined by researchers—Type-1, Type-2, Type-3, and Type-4 [11]. Type-1 (**T1**) clones are identical code fragments which may contain variations in whitespace, layouts, or comments. Type-2 (**T2**) clones are code fragments allowing variations in identifiers, literals, types, whitespace, layouts, and comments. Compared with Type-2 clones, Type-3 (**T3**) clones allow extra modifications such as changed, added, or removed statements. Type-4 (**T4**) clones further include semantically equivalent but syntactically different code fragments.

Various clone detection techniques have been built [1], [2], [3], [4], [5], [6]. For instance, SoucererCC tokenizes source code, and indexes code blocks with a token subset [5]. Given a code snippet to look for clones, SourcererCC uses the tokens from the snippet to retrieve candidate blocks, and to identify clones by matching tokens and comparing their frequencies. NiCad parses programs to pretty-print statements for code normalization, and then identifies code snippets which have similar normalized representations [3]. Both SourcererCC and NiCad mainly detect T1 and T2 clones. Deckard creates an Abstract Syntax Tree (AST) for each code fragment, and then leverages a tree similarity algorithm to identify similar code [2]. Compared with SourcererCC and NiCad, Deckard detects extra T3 clones at the cost of additional runtime overhead [5]. There is only one deep learning-based approach; it locates clones by extracting features from both program tokens and syntactic structures [12].

We have designed and implemented CCLEARNER, *the first solely token-based clone detection approach using deep learning*. We believe that the usage of terms like reserved words, type identifiers, method identifiers, and variable identifiers, can effectively characterize code implementation. If two methods use the same terms in similar ways, the methods are likely to be clones and implement similar functionalities. Meanwhile, if we consider the clone detection problem analogous to classifying code pairs as clones or non-clones, deep learning can be used to detect clones, because it can effectively find good features in training data and conduct fuzzy classification [13].

Unlike prior approaches which were specially designed to reveal certain types of clones, CCLEARNER applies deep learning on known code clones and non-clones to train a model. The model can intelligently characterize the commonality and any possible variation between clone peers regardless of the clone types. With the trained model, CCLEARNER then compares methods pair-by-pair in any given codebase to identify various kinds of clones.

Specifically, given a pair of methods (clones or non-clones), CCLEARNER uses ANTLR [14] and Eclipse ASTParser [15] to parse each method for both *usage* and *frequency* of reserved words, operators (e.g., “+”), literals, and other identifiers.

CCLEARNER then organizes the tokens into eight categories, and separately computes the similarity score for each category. Next, it characterizes the method relationship with the computed similarity vectors. For training, CCLEARNER uses both clone pairs and non-clone pairs as positive and negative examples. It extracts similarity vectors for both kinds of examples, and feeds them to a deep neural network (DNN) [16] to train a binary-class classifier. In the testing phase, CCLEARNER compares every two methods in a given codebase to predict the method relationship as "clones" or "non-clones".

We evaluated CCLEARNER and three popular clone detection tools[1] (Deckard [2], NiCad [3], and SourcererCC [5]) using BigCloneBench [7], an existing large benchmark of clone data. Compared with the other tools, CCLEARNER achieved high precision and recall with low time cost. To assess CCLEARNER's sensitivity to the parameter settings in DNN and the selected features, we experimented with different configurations of DNN, and investigated different feature sets. We observed that CCLEARNER's clone detection effectiveness varied a lot with parameter settings and selected features.

In summary, we have made the following contributions:

- We designed and implemented CCLEARNER, *the first solely token-based clone detection approach using deep learning*. With deep learning, CCLEARNER effectively captures the similar token usage patterns of clones in training data to detect clones in testing data.
- We conducted *the first systematic empirical study to compare deep learning-based clone detection with approaches not using deep learning*. CCLEARNER detected more true clones than SourcererCC and NiCad, and found clones more efficiently than Deckard.
- We investigated the sensitivity of CCLEARNER to changes in parameter settings and feature sets, and made two observations. First, CCLEARNER worked best with 2 hidden layers and 300 training iterations in DNN. Second, similarly used reserved words, markers, type identifiers, and method identifiers were good indicators of clones.

II. BACKGROUND

In this section, we first introduce some basic concepts of deep learning, and then define the terms used in the paper.

A. Concepts of Deep Learning

Deep Learning includes a set of algorithms that attempt to model high-level abstractions in data. There are various deep learning architectures, such as deep neural networks, deep belief networks, and recurrent neural networks [13]. For our project, we applied deep neural networks.

Neural Networks are a set of pattern recognition algorithms modeled after the human brain. A typical neural network system has *three* layers: input layer, hidden layer, and output layer. Each layer's output is simultaneously the subsequent layer's input. The input layer takes input data, while the output layer produces the recognized patterns [17].

[1]We contacted the authors of the existing deep learning-based clone detection tool [12], but were unable to get the tool.

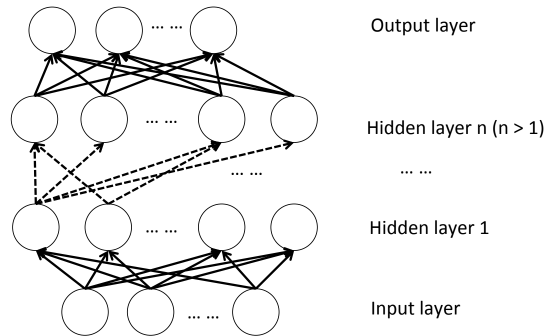

Fig. 1: The deep neural network architecture with one input layer, one output layer, and multiple hidden layers.

Deep Neural Networks (DNNs) are neural networks with *multiple* hidden layers between the input and output layers [18]. As shown in Fig. 1, a DNN has at least two hidden layers to process data in multi-steps for pattern recognition. Each node in the network combines its inputs, such as x_1, x_2, $\ldots$, x_m, with a set of *coefficients* or *weights* to either amplify or dampen the inputs. In this way, each node separately determines which input is most helpful for the overall learning task (such as classifying data), and whether or to what extent each input progresses further through the network to affect the ultimate outcome, say, an act of classification [16].

It is challenging to decide the *best* configuration of coefficients or weights in one trial, so a DNN usually involves *multiple iterations* to tune those coefficients. In the initial iteration, a DNN makes the initial guess for coefficients of all nodes, predicts labels accordingly, and then measures the prediction error against the ground truth labels. This error measurement provides feedback to the next iteration for coefficient adjustment. Therefore, even though the initial iteration may produce a bad prediction model with coefficients poorly guessed, as more iterations take place, the model gradually evolves to a better one with coefficients updated and the prediction error minimized.

Like other machine learning techniques, DNN can also be classified into two types: supervised and unsupervised learning. The former is usually used in classification tasks that depend upon labeled data sets. The latter is widely applied to pattern analysis and clustering, both of which require unlabeled data. DNN has performed well in many classification tasks, such as face detection, objects identification, gestures recognition and spam classification [16].

B. Terminologies

In this paper, a **clone method pair** or **true clone pair** includes two methods with similar code. Each method in the clone method pair is called a **clone peer** of the other method. A **non-clone method pair** or **false clone pair** includes two methods implemented with dissimilar code, and each of the methods is called a **non-clone peer** of the other.

III. APPROACH

We designed and implemented CCLEARNER to detect clones with token usage analysis and deep learning. As shown

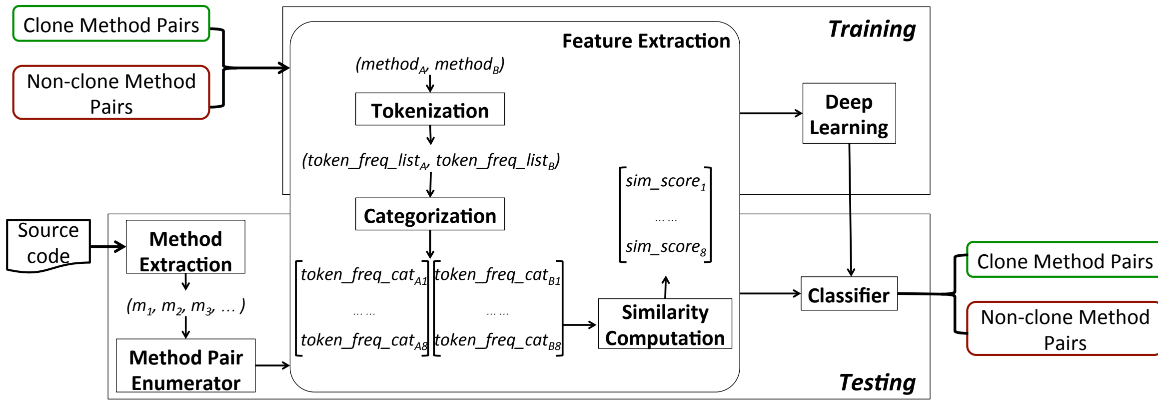

Fig. 2: CCLearner consists of two phases: training and testing. The first phase takes in both clones and non-clones to train a classifier in a deep learning framework. The second phase takes in a codebase to detect clones with the trained classifier.

in Fig. 2, there are training and testing phases in our approach. Both phases extract features of token usage to either characterize or detect clones. In this section, we will first discuss how features are extracted (Section III-A), and then explain the training and testing phases (Section III-B and III-C).

A. Feature Extraction

To extract the features that characterize the clone (or non-clone) relationship of method pairs, for each given method pair $(method_A, method_B)$, CCLEARNER first tokenizes code using ANTLR lexer [14] to identify all tokens used in each method, and then records the occurrence count of each token. Such token usage information is organized as a *token-frequency list* for each method, represented as $token_freq_list_A$ for $method_A$ and $token_freq_list_B$ for $method_B$ in Fig. 2. CCLEARNER then categorizes tokens and computes features as similarity scores of token usage.

Token Categorization and Extraction. When various tokens are extracted from a method, such as reserved words (e.g., "`for`" and "`if`") and operators (e.g., "`+`" and "`&`"), *different types of tokens may have different capabilities to characterize clones*. For instance, T1-T3 clones are more likely to have common reserved words than common operators, because these clones almost always have identical program syntactic structures but may contain slightly different arithmetic or logic operations. Therefore, CCLEARNER classifies tokens into eight categories to create eight disparate features. Table I presents the eight token categories with an exemplar token-frequency list shown for each category.

TABLE I: Token categories

Index	Category name	An exemplar token-frequency list
C1	Reserved words	<if, 2>, <new, 3>, <try 2>, ...
C2	Operators	<+=, 2>, <!=, 3>, ...
C3	Markers	<;, 2>, <[, 2>, <], 2>, ...
C4	Literals	<1.3, 2> , <false, 3>, <null, 5>, ...
C5	Type identifiers	<byte, 2> , <URLConnection, 1> ,...
C6	Method identifiers	<read, 2> , <openConnection, 1>, ...
C7	Qualified names	<System.out, 6>, <arr.length, 1>, ...
C8	Variable identifiers	<conn, 2> , <numRead, 4>, ...

In Table I, we use a list of <key, value> pairs to represent each token-frequency list, where "key" is a token, and "value" shows the occurrence count. The token-frequency lists of C1-C3 can be easily constructed from ANTLR lexer output, because the token sets of reserved words, operators, and markers are well defined. However, for C4-C8 tokens, ANTLR cannot always precisely identify tokens or clearly differentiate between various types of tokens. For instance, given positive numbers and negative numbers (e.g. `1` and `-1`), ANTLR only extracts positive numbers (e.g., `1`) as literals, although negative numbers should also be treated as literals. Furthermore, given an identifier `foo`, ANTLR cannot tell whether the identifier represents a type or a variable.

To precisely identify C4-C8 tokens, CCLEARNER also uses Eclipse ASTParser [15] to create an Abstract Syntax Tree (AST) for each method, and implements separate ASTVisitors to extract different kinds of tokens by visiting certain types of AST nodes. For example, given a statement `int v = -1 + a.b.foo()`, CCLEARNER creates the AST in Fig. 3. Notice that ASTParser does not replace ANTLR in CCLEARNER, because ASTParser cannot reveal all tokens that ANTLR detects, including reserved words and markers.

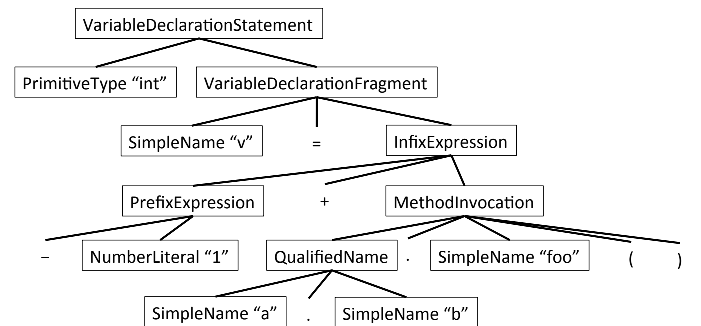

Fig. 3: The AST of `int v = -1 + a.b.foo()`

To identify literals (C4 tokens), an ASTVisitor [19] was implemented to visit all literal AST nodes (e.g., BooleanLiteral and CharacterLiteral), and to create an entry in the token-frequency list for each unique literal. For the AST in Fig. 3, when the NumberLiteral (`1`) is visited, CCLEARNER further checks whether the literal is a subexpression of a negative

number (i.e. `-1`); if so, the negative number is extracted. In this way, we differentiate between positive and negative numbers.

For type identifier (C5 token) extraction, we implemented an ASTVisitor to visit all type-relevant AST nodes like PrimitiveType, TypeLiteral, etc.. For our example, when PrimitiveType (`int`) is accessed, CCLEARNER creates an entry for the type name, and then counts the name's occurrence.

To extract method identifiers (C6 tokens), we built an ASTVisitor to visit all method-relevant AST nodes such as MethodDeclaration and MethodInvocation. In this way, all method names used in a method implementation can be precisely identified. In our example, there is a MethodInvocation element, and CCLEARNER thus extracts the method name `foo`.

Qualified names (C7 tokens) are special tokens, so we created an ASTVisitor to especially process all QualifiedName nodes. Given a qualified name like `Foo.Goo.Bar`, we may interpret it as a type identifier or variable identifier, we can also interpret part of the name like `Foo.Goo` as a type or variable. With the limited program syntactic information derived from a method's AST, we cannot always precisely decide whether a qualified name represents a type. Although we can invoke the resolveBinding() API on a qualified name, the invocation does not always help resolve the name binding, either. To avoid any inaccurate categorization of type or variable identifiers, we thus simply treat qualified names as a separate category of tokens. For the example in Fig. 3, `a.b` is extracted as a qualified name, even though we do not know whether it represents a type or a variable.

Variable identifiers (C8 tokens) are harder to extract than above tokens, because there is no fixed set of AST node types which may contain such identifiers in particular positions. Therefore, we created an ASTVisitor to visit all SimpleName nodes, and to deduct the SimpleNames that are covered by tokens belonging to any of above categories. In our exemplar AST, there are four SimpleName elements: `v`, `a`, `b`, and `foo`. Since `a` and `b` are covered by the qualified name `a.b`, while `foo` is already identified as a method name, CCLEARNER ends up with recognizing one variable identifier (`v`) in this way.

In summary, as shown in Fig. 2, after extracting different categorized tokens for each method under comparison, CCLEARNER creates vectors of token-frequency lists, which are represented as $[token_freq_cat_{A1}, \ldots, token_freq_cat_{A8}]$ for $method_A$, and $[token_freq_cat_{B1}, \ldots, token_freq_cat_{B8}]$ for $method_B$.

Similarity Computation. When two methods are characterized as vectors of token-frequency lists, we believe *the similarity between the vectors should reflect the similarity of their methods*. Intuitively, the more similar two vectors are, the more likely those two methods are clones to each other. Correspondingly, if two methods are dissimilar, their vectors are divergent. Therefore, for each method pair, CCLEARNER computes a similarity score between the token-frequency lists of each token category, and then constructs an eight-value similarity vector to characterize the method pair relationship.

Formally, for category $C_i (1 \leq i \leq 8)$, if we represent the two term-frequency lists as L_{Ai} and L_{Bi}, then the similarity score sim_score_i is calculated as below:

$$sim_score_i = 1 - \frac{\sum_x |freq(L_{Ai}, x) - freq(L_{Bi}, x)|}{\sum_x (freq(L_{Ai}, x) + freq(L_{Bi}, x))}, \quad where\ x \in tokens(L_{Ai}) \cup tokens(L_{Bi}). \quad (1)$$

Intuitively, we enumerate all tokens contained in either of the two lists. For each enumerated token x, we compute both its absolute frequency difference value $(|freq(L_{Ai}, x) - freq(L_{Bi}, x)|)$ and frequency sum $(freq(L_{Ai}, x) + freq(L_{Bi}, x))$ between the lists. Next, we sum up all absolute frequency difference values and all frequency summations separately to compute a ratio, which is deducted from 1 to get the similarity score. According to the formula, the range of similarity score is $[0, 1]$.

Our similarity calculation formula is meaningful. When two methods are identical and have purely identical token-frequency lists, the ratio between the absolute frequency difference sum and total frequency sum is always 0, so the similarity score for each category is 1. When two methods are totally different and share no token in common, the corresponding ratio is 1, while the similarity score is 0.

Suppose we have two token-frequency lists: $L_A = \{<a, 3>, <b, 4>, <c, 5>\}$ and $L_B = \{<b, 3>, <c, 6>, <d, 2>\}$. The similarity score is computed as $1-(|3-0|+|4-3|+|5-6|+|0-2|)/((3+0)+(4+3)+(5+6)+(0+2)) = 0.70$. Generally speaking, the more tokens shared between lists and the less frequency difference there is for each token, the higher similarity score we can derive. Note that if two methods have no token for certain category (such as keywords), we set the corresponding similarity score as 0.5 by default. We tried to set the default value as 0 or 1, but none of these values worked as well as 0.5. Maybe the 0.5 works because it makes the similarity in that token category to be equally meaningful to determining clone pairs or non-clone pairs.

With the features extracted as similarity vectors, CCLEARNER further leverages deep learning to train and test a classifier for clone detection.

B. Training

To train a binary-class classifier for clone detection, we need feature data for both positive and negative examples of the clone relationship. The positive examples are feature vectors extracted from clone method pairs, while the negative examples are vectors derived from non-clone pairs. In our training data, each data point has the following format: $< similarity_vector, label >$, where $similarity_vector$ is a vector of eight similarity scores, and $label$ is either 1 to represent "$CLONE$", or 0 to represent "NON_CLONE".

As our approach is built on the token-frequency list comparison between methods, when method bodies are small, any minor variation of token usage can cause significant degradation of similarity scores, making the training data noisy. To avoid any confusion caused by small clone methods,

we intentionally constructed training data with methods that contained at least six lines of code.

We used DeepLearning4j [20]—an open source library of DNN—to train a classifier. Since there are eight features defined, we configured eight nodes for the input layer, with each node independently taking in one feature value. As there are two labels for the classification task: "$CLONE$" and "NON_CLONE", the output layer contains two nodes to present DNN's classification result. CCLEARNER configures the DNN to include 2 hidden layers and to run 300 iterations for training, because our experiments in Section IV-E show that CCLEARNER worked best with these parameter settings.

Research literature recommended that the number of nodes in each hidden layer (h_num) should be less than twice of the number of input nodes (i_num) [21]. In CCLEARNER, since $i_num = 8$, we simply picked $h_num = 10$ as the default setting so that $h_num < 2 * i_num$.

C. Testing

Given a code base, CCLEARNER first extracts methods from source code files with Eclipse ASTParser, and then enumerates all possible method pairs. CCLEARNER feeds each enumerated method pair to the trained classifier to decide whether the methods are clones. If two methods are judged as clones, CCLEARNER reports them accordingly.

Theoretically, when n methods are extracted from a code base, there are $n(n-1)$ pairs enumerated, and the clone detection algorithm complexity can be $O(n^2)$. To reduce the comparison run-time overhead, we implemented two heuristics to filter out some unnecessary comparisons between methods. One filter was designed to compare two methods' lines of code (LOC). If one method's LOC is more than three times of the other method's LOC, it is very unlikely that the methods are clones, and we can simply conclude that the methods are non-clones, skipping any further processing to extract features or execute the classifier. Another filter removes any candidate method with less than six LOC for two reasons. First, small methods may contain so few tokens that CCLEARNER cannot effectively detect them based on the token usage comparison. Second, prior research shows that the six-line minimum is common in clone detection benchmarking [11], [5].

The output layer has two nodes to separately predict the likelihood of clones and non-clones: l_c and l_{nc}, where $l_c + l_{nc} = 1$. In order to map the continuous likelihood values to either "CLONE" or "NON_CLONE" discrete classification, we require $l_c \geq 0.98$ to precisely detect clones without producing many false alarms.

IV. EVALUATION

In this section, we will first describe the benchmark we used (Section IV-A), and how we constructed training and testing data sets with the benchmark (Section IV-B). We then describe the metrics and how they were calculated to evaluate the effectiveness of clone detection approaches (Section IV-C). With the metrics defined, we compared CCLEARNER with three popular approaches that detect clones without using machine learning or deep learning (Section IV-D). Finally, we investigated different parameter settings (Section IV-E) and various feature sets (Section IV-F) to explore the best way of applying deep learning in CCLEARNER.

A. Benchmark

We reused BigCloneBench [7], an existing large benchmark of code clones. Svajlenko et al. built the benchmark by mining for clones of specific functionalities, and manually labeling clones and non-clones. After downloading the benchmark tar file from its website [22], we leveraged PostgreSQL [23] to load the benchmark data. The database covers implementations of 10 functionalities, containing over 6 million tagged true clone pairs and 260 thousand tagged false clone pairs. The 10 functionalities correspond to 10 source code folders indexed as **Folder #2-#11** by the benchmark creators. The alternative implementations of each functionality are put into the source code files of one folder. Among the data, the true clones are tagged as **T1**, **T2**, **VST3 (Very Strong Type 3)**, **ST3 (Strong Type 3)**, **MT3 (Moderately Type 3)**, or **WT3/4 (Weak Type 3 or Type 4)** [24]. In our experiment, the downloaded latest version of BigCloneBench has slightly different numbers of clones in the database from the numbers reported in the paper, so we presented our observed numbers in Table II.

As shown in the table, true clone pairs do not evenly distribute among the 10 folders. Specifically, **Folder #4** has 22,113 source files, 4,676,552 LOC, containing the largest number of known true clone pairs and false clone pairs. In contrast, **Folder #5** only covers 56 source files, 3,527 LOC, and contains the fewest true and false clone pairs.

B. Data Set Construction

To evaluate CCLEARNER, we need a training data set to create a classifier, and a testing data set to assess the classifier. In Table II, since **Folder #4** has the largest number of both true and false clone pairs, we used the data in this folder for training, and the data in other folders for testing. Table III shows details of both data sets.

In Table III, the number of true clones and false clones used for training are smaller than the corresponding numbers of **Folder #4** in Table II. There are three reasons to explain the difference. First, small methods may contain so few tokens that the resulting token-frequency lists may vary significantly even though their methods are very similar. To avoid any noise caused by small clone methods, we intentionally filtered out methods containing less than six LOC to construct training data. Second, MT3 and WT3/4 clones may contain totally different implementations of the same functionality. Training a classifier with such noisy data can cause the resulting classifier to wrongly report a lot of clones and to produce many false alarms. Therefore, we excluded MT3 and WT3/4 clones from the training data. Third, we randomly chose a subset of false clone pairs from **Folder #4** to achieve a count balance between the positive examples and negative examples. In other words, we ensured that the total number of false clones is equal to

TABLE II: Data in the downloaded BigCloneBench

Folder Id.	# of Source Files	LOC	# of True Clone Pairs						# of False Clone Pairs
			T1	T2	VST3	ST3	MT3	WT3/4	
#2	10,372	1,984,327	1,553	9	22	1,412	2,689	404,277	38,139
#3	4,600	812,629	632	587	525	2,760	24,621	862,652	4,499
#4	22,113	4,676,552	13,802	3,116	1,210	4,666	23,693	4,618,462	197,394
#5	56	3,527	0	0	0	0	1	34	12
#6	472	83,068	4	0	14	50	124	24,338	4,147
#7	1,037	299,525	39	4	21	212	1,658	11,927	15,162
#8	131	18,527	3	7	5	0	2	259	78
#9	669	107,832	0	0	0	0	0	55	1,272
#10	1,014	286,416	152	64	285	925	2,318	236,726	1,762
#11	64	6,736	0	0	1	6	0	245	0
Total	**40,528**	**8,279,139**	**16,185**	**3,787**	**2,083**	**10,031**	**55,106**	**6,158,975**	**262,465**

TABLE III: Data sets of training and testing in CCLEARNER

Data set	# of True Clone Pairs						# of False Clone Pairs
	T1	T2	VST3	ST3	MT3	WT3/4	
Training	13,750	3,104	1,207	4,602	0	0	22,663
Testing	2,383	671	873	5,365	31,413	1,540,513	0

the total number of true clones (the sum of T1, T2, VST3, and ST3 clones), which is 22,663.

To construct the testing data, we included all source files in the other 9 folders (except **Folder #4**) of BigCloneBench. CCLEARNER automatically extracts methods from these files, and compares methods pair-by-pair for clone detection. Since our evaluation focus is *whether* CCLEARNER *can correctly identify clone method pairs*, the testing data oracle only includes the known true clone pairs that we expect CCLEARNER to retrieve, without covering any known false clone pair.

C. Metrics and Their Calculations

We defined and calculated the following metrics to evaluate the effectiveness of any clone detection approach:

Recall (R) measures among all known true clone pairs, how many of them are detected by a clone detection approach:

$$R = \frac{\text{\# of true clone pairs detected}}{\text{Total \# of known true clone pairs}}. \quad (2)$$

Given a clone detection tool such as CCLEARNER, we could automatically evaluate the recall for individual clone types or for some clone types (T1-T4) as a whole. To get the per-clone-type recall rate of CCLEARNER, we first derived the set intersection between CCLEARNER's retrieved clones and the known clones in BigCloneBench for each clone type, and then computed the ratio between the interaction set and the known clones. Since many clone detection tools cannot effectively retrieve MT3 and WT3/4 clones, similar to prior work [5], we evaluated the overall recall for T1, T2, VST3, and ST3 typed clones as below:

$$R_{T1-ST3} = \frac{\text{\# of true clone pairs (of T1-ST3)}}{\text{Total \# of known true clone (of T1-ST3)}}. \quad (3)$$

Precision (P) measures among all of the clone pairs reported by a clone detection approach, how many of them are actually true clone pairs:

$$P = \frac{\text{\# of true clone pairs}}{\text{Total \# of detected clone pairs}}. \quad (4)$$

Given a reported set of clones, it would be ideal to intersect this set with the tagged true clone set in BigCloneBench to compute precision. However, in reality, such intersection approach is infeasible because according to our experience, BigCloneBench tags only an incomplete set of the actual true clones. We observed untagged true clones. With such incomplete ground truth of clone pairs, we cannot decide automatically whether a reported clone pair is true or false if the pair is not covered by the incomplete set.

To properly evaluate the precision of CCLEARNER and other approaches, we randomly sampled each tool's reported clone pairs, and manually examined the method pairs to check whether they were true clones. To ensure that our sampled data is representative, we chose 385 reported clones for each approach. When an approach reports hundreds of thousands of clone pairs, 385 is a statistically significant sample size with a 95% confidence level and ±5% confidence interval.

With such sampling-based precision evaluation approach, we cannot control how many clones are sampled for each type. Therefore, we could not evaluate the precision rate per type. Instead, we estimated the overall precision as below:

$$P_{estimated} = \frac{\text{\# of true clone pairs}}{\text{385 detected clone pair samples}}. \quad (5)$$

C score (C) combines $P_{estimated}$ and R_{T1-ST3} to measure the overall accuracy of clone detection as below:

$$C = \frac{2 * P_{estimated} * R_{T1-ST3}}{P_{estimated} + R_{T1-ST3}}. \quad (6)$$

C score varies within [0, 1]. The higher C scores are desirable, because they demonstrate better accuracy at clone detection. Suppose we have 100 T1-ST3 true clone pairs in a codebase, and CCLEARNER reports 120 clone pairs, with 80 of them being true clones. Therefore, $R_{T1-ST3} = 80/100 = 80\%$, as 80 out of the 100 true clone pairs are retrieved. $P_{estimated} = 80/120 = 67\%$, because among the

120 reported clone pairs, only 80 pairs are correctly identified. $C = 2 * 80\% * 67\%/(80\% + 67\%) = 73\%$.

D. Effectiveness Comparison of Clone Detection Approaches

To evaluate CCLEARNER's effectiveness of clone detection, we compared our approach with three popular clone detection tools: SourcererCC [5], NiCad [3], and Deckard [2]. All these three tools were executed with the default parameter configuration on CCLEARNER's testing data. As with CCLEARNER, SourcererCC detects clones based on the tokens extracted from source code. NiCad normalizes the code and compares the code line by line. Deckard compares the ASTs of code to report clones with similar tree structures. We chose these tools because they are widely used, and can well represent the mainstream types of clone detection approaches: token-based and tree-based.

We compared all tools in four aspects: *recall*, *precision*, *C score*, and *time cost*.

TABLE IV: Recall comparison among tools (%)

	T1	T2	VST3	ST3	MT3	WT3/4
CCLearner	100	98	98	89	28	1
SourcererCC	100	97	92	67	5	0
NiCad	100	85	98	77	0	0
Deckard	96	82	78	78	69	53

Recall. As shown in Table IV, CCLEARNER worked better than SourcererCC and NiCad. Compared with Deckard, CCLEARNER effectively detected more T1-ST3 clones, but failed to report many of the MT3 and WT3/4 clones.

There are two reasons to explain CCLEARNER's insufficiency. First, CCLEARNER relies on the exactly same terms used in different methods to compute similarity vectors. When two clone methods share few identifiers and contain significantly divergent program structures, CCLEARNER cannot detect the clone relationship. Second, according to the BigCloneBench paper [7], within each WT3/T4 clone pair, the two methods share less than 50% of statements in common. Without reasoning about the semantics of syntactically different but semantically equivalent code snippets, it is very challenging for any token-based approach to detect such clones. In the future, we plan to devise some supplementary techniques for these specialist clones.

It is easy to explain why Deckard detected more MT3 and WT3/4 clones than all token-based approaches. Deckard compares *program syntactic structures* instead of *tokens* to detect code similarity, so the tool is more robust to significant variations of token usage between clones. If a method does not share many tokens with its clone method, or organizes tokens in a different sequential order from its clone, Deckard is still capable of identifying the structural similarity. However, it is difficult to understand why Deckard missed some simple clones (of T1 or T2). Figure 4 shows a clone method pair of T2 in BigCloneBench, where differently used terms are **bolded**. Deckard failed to detect this clone pair maybe due to some implementation issue.

```
1. public static String MD5(String text) throws
2.     NoSuchAlgorithmException,
3.     UnsupportedEncodingException {
4.   MessageDigest md;
5.   md = MessageDigest.getInstance("MD5");
6.   byte[] md5hash = new byte[32];
7.   md.update(text.getBytes("iso-8859-1"),0,text.length());
8.   md5hash = md.digest();
9.   return convertToHex(md5hash);
10.  }
11.
12.public static String SHA1(String text) throws
13.    NoSuchAlgorithmException,
14.    UnsupportedEncodingException {
15.  MessageDigest md;
16.  md = MessageDigest.getInstance("SHA-1");
17.  byte[] sha1hash = new byte[40];
18.  md.update(text.getBytes("iso-8859-1"),0,text.length());
19.  sha1hash = md.digest();
20.  return convertToHex(sha1hash);
21.  }
```

Fig. 4: A clone method pair of T2 in BigCloneBench

We found that NiCad could not detect the above T2 clone pair, either. This may be because NiCad does not tolerate any literal difference between clones when matching program statements. Specifically, among the 6 statements in each method (line 4-9 and line 15-20), there are 2 statements using divergent literals ("`MD`" vs. "`SHA-1`" in line 5 and 16, and "`32`" vs. "`40`" in line 6 and 17). Therefore, the similarity between the two methods is calculated as $4/6 = 67\%$, which is below NiCad's default similarity threshold 70%. As a result, the two methods were wrongly judged as non-clones.

TABLE V: The sampled precision and the total number of reported and true clone pairs of each tool

	CCLearner	SourcererCC	NiCad	Deckard
$P_{estimated}$(%)	93	98	68	71
# of reported clone pairs	548,987	265,611	646,058	2,301,526
# of estimated true clone pairs	510,558	260,299	439,319	1,634,083

Precision. Table V shows the $P_{estimated}$, the number of reported clone pairs, and the number of estimated *true* clone pairs for each approach. We calculated the number of *true* clones by multiplying each $P_{estimated}$ by the corresponding number of reported clones. Compared with SourcererCC and NiCad, CCLEARNER reported the most true clone pairs. Deckard had a lower $P_{estimated}$, but reported the most true clone pairs over all approaches.

We looked more closely at SourcererCC which had the highest $P_{estimated}$, but reported the fewest true clone pairs. One reason for this may be SourcereCC's partial index filtering. SourcererCC indexes methods with *only the most rare tokens* used in each method, and detects clones *purely among the methods indexed with the same tokens*. If two clone methods' most rare identifiers are different, SourcererCC may index them differently, and wrongly exclude them from further similarity comparison.

C Score. Table VI shows the C score comparison between tools. CCLEARNER detected clones more accurately than other tools because its C score was the highest, which means that CCLEARNER had both high estimated precision and high T1-ST3 recall.

TABLE VI: C score comparison between tools (%)

CCLearner	SourcererCC	NiCad	Deckard
93	88	76	77

Time Cost. When comparing clone detection tools, time cost is also an important factor to consider. This is because if a tool spends too much time detecting clones, it may not respond quickly enough to users' requests, neither will it scale up to large codebases containing millions of lines of code.

TABLE VII: Time cost of tools on about 3.6M LOC

CCLearner	SourcererCC	NiCad	Deckard
47m	13m15s	33m40s	4h24m

As shown in Table VII, when detecting clones in a codebase with 3,602,587 LOC, SoucererCC ran the fastest, taking only 13 minutes 15 seconds. NiCad was slower than SourcererCC and took 33 minutes 40 seconds. Deckard worked the most slowly and spent 4 hours 24 minutes, because it used an expensive tree matching algorithm. Our observations on the three tools' time cost comparison align with the findings in prior work [5].

CCLEARNER detected clones within 47 minutes. Similar to SourcererCC and NiCad, CCLEARNER worked faster than Deckard because it did not reason about program structures. However, CCLEARNER was slower than NiCad and SourcererCC, because it extracted features from method pairs, trained a classifier before detecting any clone, and compared methods pair-by-pair to find clones.

In addition to the 47 minutes, CCLEARNER spent 5 minutes on training. As a classifier can be applied to different codebases once it is trained, the time spent on training could be considered as one-time cost. Due to the pair-by-pair method comparison mechanism, CCLEARNER's clone detection is an embarrassingly parallel task [25], which means that we can easily parallelize the task in the future to further reduce CCLEARNER's time cost.

In summary, CCLEARNER effectively detects various types of clones with high precision, high recall, and low time cost.

E. Parameter Settings for Training a DNN Model

Two parameters in the DNN algorithm may affect CCLEARNER's effectiveness: the number of hidden layers ($layer_num$) and the number of iterations ($iter_num$). To properly configure these parameters, we have experimented with different parameter values to investigate the best settings.

As shown in Table VIII, we tried different numbers of hidden layers in CCLEARNER: 2, 4, 6, 8, and 10, setting $iter_num = 200$ by default. We found that with 8 or 10 hidden layers, the weights or coefficients used in DNN were not converging, and the trained models were unusable. When $layer_num = 2$, CCLEARNER achieved the highest C score.

We also experimented with different numbers of iterations in CCLEARNER: 100, 200, 300, 400, and 500, with $layer_num = 2$ by default. As shown in Table IX, when $iter_num = 300$, CCLEARNER worked best. Therefore, we used $layer_num = 2$ and $iter_num = 300$ as the default setting to conduct other experiments.

F. Sensitivity to Feature Selection

There are eight features used in CCLEARNER, but we do not know which feature is most important, or how different features affect the overall clone detection effectiveness. Therefore, we investigated eight variant feature sets, ran CCLEARNER with each feature set, and measured the recall, precision, and time cost. To create a variant set, we removed one feature from the original set at a time, and used the remaining seven features to characterize any method relationship. As shown in Table X, **Set 0** corresponds to the original feature set, while **Sets 1-8** are the variant sets generated in the above way.

Based on Table X, we made three interesting observations. First, *all feature sets achieved the same or very close recall when detecting T1, T2, and VST3 clones*. This may be because the peers of such clones were very similar to each other and used the majority of tokens in the same way. Even though one kind of tokens were ignored when extracting features, the clone detection results were not affected.

Second, compared with the original feature set (**Set 0**), **Sets 1**, **3**, **5** and **6** acquired both lower precision and lower recall. This means that in **Set 0**, *reserved words, markers, type identifiers, and method identifiers were important features, and clone peers were likely to share many of such terms*. Two possible reasons can explain this finding. First, according to the BigCloneBench paper [7], T1-MT3 clones have at least 50% syntactic similarity between method peers, so such clones may share many structure-relevant reserved words like `while`, `switch`, etc., and many markers to indicate similar statement formats, such as"`[`" and "`:`". Second, when clone methods implement the same semantics, they seldom use totally different types or irrelevant methods for data storage or numeric computation.

Third, compared with **Sets 2** and **7**, the original feature set achieved higher recall but maintained the same precision. It means that both true and false clone pairs may use operators and qualified names similarly. These two kinds of terms can help retrieve true clones, but may not filter out false clones as effectively as the four types of terms mentioned above.

V. THREATS TO VALIDITY

In our evaluation, we intersected the clone pairs found by CCLEARNER with the tagged clone pairs in BigCloneBench to automatically evaluate recall. However, based on our observation, BigCloneBench did not tag all clones actually existing in the codebase. Instead, it tagged a subset of the actual clones. Evaluating CCLEARNER against such partial ground truth may cause the measured recall rates different from the actual rates.

We evaluated CCLEARNER's precision by having three authors separately examine sample sets of reported clones. Therefore, the measured precision rates may be subject to our human bias or unintentional errors. In the future, we would like to crowdsource the task or recruit external developers to

TABLE VIII: CCLEARNER's effectiveness with different hidden layer numbers

Hidden Layer #	R% **Per Type**						$R_{T1-ST3}\%$	$P_{estimated}\%$	$C\%$
	T1	**T2**	**VST3**	**ST3**	**MT3**	**WT3/4**			
2	100	98	98	83	16	0	89	95	92
4	100	98	98	92	37	1	95	87	91
6	100	98	98	94	47	2	96	69	80
8	Not converging								
10	Not converging								

TABLE IX: CCLEARNER's effectiveness with different iteration numbers

Iteration #	R% **Per Type**						$R_{T1-ST3}\%$	$P_{estimated}\%$	$C\%$
	T1	**T2**	**VST3**	**ST3**	**MT3**	**WT3/4**			
100	100	98	94	68	4	0	81	100	89
200	100	98	98	83	16	0	89	95	92
300	100	98	98	89	28	1	93	93	93
400	100	98	98	91	38	1	95	84	89
500	100	98	98	93	45	2	95	77	85

TABLE X: CCLEARNER's effectiveness with different feature sets

No.	Feature Set Description	R% **Per Type**						$R_{T1-ST3}\%$	$P_{estimated}\%$	$C\%$
		T1	**T2**	**VST3**	**ST3**	**MT3**	**WT3/4**			
0	All eight features	100	98	98	89	28	1	93	93	93
1	Without C1 tokens (reserved words)	100	98	97	82	16	0	89	91	90
2	Without C2 tokens (operators)	100	98	98	84	18	0	90	93	91
3	Without C3 tokens (markers)	100	98	98	83	18	0	90	91	91
4	Without C4 tokens (literals)	100	98	97	87	25	1	92	87	89
5	Without C5 tokens (type identifiers)	100	98	98	83	16	0	90	86	88
6	Without C6 tokens (method identifiers)	100	98	98	86	20	1	92	77	84
7	Without C7 tokens (qualified names)	100	98	98	85	21	0	91	93	92
8	Without C8 tokens (variable identifiers)	100	98	97	87	24	1	92	83	87

manually examine the reported clones. We will ensure that each clone is checked by at least two people to avoid some random errors.

Limited by the partial ground truth data in BigCloneBench, we defined our own metric (C score), instead of reusing the well known F score (harmonic mean of precision and recall), to assess the overall accuracy of clone detection techniques. This newly defined metric may bias our parameter tuning and feature selection for CCLEARNER, and the comparison experiment between different clone detection tools.

White et al. built a clone detection tool using deep learning [12]. We were not able to get the tool to empirically compare it with CCLEARNER. Therefore, we do not know how our deep learning-based approach compares to theirs.

VI. RELATED WORK

This section describes related work on clone detection techniques, search-based software engineering in clone detection optimization, and the application of deep learning in software engineering research.

Clone Detection. There are mainly five types of clone detection techniques developed: text-based, token-based, tree-based, graph-based, and metrics-based [26].

Text-based clone detection techniques use line-based string matching algorithms to detect exact or near-duplication clones [27], [28]. These approaches are simple and fast, however, they mainly locate T1 clones and cannot detect more complicated clones.

Token-based clone detection tools first identify tokens and remove white spaces and comments from source code. They then detect clones based on token comparison [1], [29], [3], [5]. For instance, CCFinder and NiCad replace identifiers related to types, variables, and constants with special tokens, and then compare the resulting token sequences [1], [3]. CP-Miner further converts the token sequence of each statement to a hash code, concisely representing a program as a sequence of hash codes. It then cuts the number sequence into smaller sub-sequences to efficiently detect similar code [29]. SourcererCC indexes code blocks with the least frequent tokens contained in the blocks, and then compares blocks indexed by the same token to find clones [5]. Compared with text-based clone detection, token-based approaches are robust to formatting and renaming changes, and can handle T2 clones.

Tree-based clone detection creates an Abstract Syntax Tree (AST) for each code fragment, and then leverages tree matching algorithms to detect similar subtrees [30], [2]. For example, Baxter et al. hashed subtrees to B buckets, and only compared subtrees in the same bucket to avoid unnecessary comparisons between dissimilar code [30]. Deckard represents each subtree as a counter vector of different categories of tokens, and then clusters and compares subtrees based on those vectors [2]. Since tree-based tools care about program syntactic structures, they can tolerate variations in the number of statements, and thus can find near-miss clones. However, prior work showed that such approaches might not scale well to large codebases due to the significant time cost and memory usage [5].

Graph-based clone detection leverages program static analysis to construct a Control Flow Graph (CFG) [31] or Program Dependence Graph (PDG) [32] for each code fragment, and then uses subgraph matching algorithms to find similar code [6], [33], [34], [35]. For instance, Apiwattanapong et al. defined and used enhanced control flow graphs to match Object-oriented programs, and to decide what is the difference between the two versions of a modified program [6]. Krinke and Liu et al. leveraged PDGs [34], [35], while Komondoor et al. further sliced programs on PDGs [33] to detect clones. As CFGs and PDGs can present program semantics while abstracting away low-level details like token usage, graph-based clone detection is capable of detecting near-miss clones. However, the subgraph matching algorithms are usually so expensive that the approaches are not scalable, and we could not find any such tool usable for our tool comparison evaluation.

Metrics-based clone detection gathers different metrics of code fragments, and compares the metric value vectors to identify clones [36], [37]. The collected metrics may include the number of function calls, the number of declaration statements, cyclomatic complexity, and the ratio of input/output variables to the fanout. Such approaches do not leverage any concrete program information like tokens or program structures for code characterization. No such tool is available for use.

Search-Based Software Engineering (SBSE) in Clone Detection Optimization. Two approaches used SBSE to find a set of parameter values that maximize the agreement between an ensemble of clone detection tools [38], [39]. Specifically, Wang et al. presented EvaClone, an approach using a Genetic Algorithm (GA) to search the configuration space of clone detection tools, and to find the best parameter settings [38]. Ragkhitwetsagul et al. [39] did a replication study on EvaClone, and observed that the optimized parameters outperform the tools' default parameters in term of clone agreement by 19.91% to 66.43%. However, EvaClone gives undesirable results in terms of clone quality. With deep learning, we aim to identify the best usage of similarity vectors when detecting clones, instead of maximizing the agreement of different clone detection tools.

Deep Learning-Based Research in Software Engineering. Researchers have recently used deep learning to solve problems in Software Engineering [40], [41], [42], [12]. For instance, Lam et al. combined deep learning with information retrieval to localize buggy files based on bug reports [40]. Wang et al. used deep belief network [43] to predict defective code regions [41]. Gu et al. used deep learning to generate API usage sequences for a given natural language query [42].

White et al. recently presented a deep learning-based clone detection tool [12]. The tool first used recurrent neural network [44] to map program tokens to continuous-valued vectors, and then used recursive neural network [45] to combine the vectors with extracted syntactic features to train a classifier. However, they did not compare their approach with any existing clone detection technique using any well-known clone benchmark. Different from White et al.'s work, CCLEARNER directly extracts features based on different categories of tokens in source code, and purely relies on similar token usage to detect clones. We conducted a comprehensive comparison between CCLEARNER and three popular approaches which detect clones without deep learning. We used the BigCloneBench data set in our experiment for a scientific tool comparison. Since White et al.'s tool is not available even though we contacted the authors, we could not compare CCLEARNER with it empirically.

VII. CONCLUSION

We presented CCLEARNER, a deep learning-based clone detection approach. Compared with most prior work, CCLEARNER does not implement specialized algorithms to target certain types of clones. Instead, it infers the commonality and possible variation patterns between the peers of known clone method pairs, and then further identifies method pairs matching the patterns to report clones. CCLEARNER is the first solely token-based clone detection approach using deep learning.

With the BigCloneBench data, we conducted the first systematic empirical study to compare deep learning-based clone detection with other approaches that do not use deep learning. Compared with existing token-based approaches, CCLEARNER detected more diverse clones with high precision and recall. Compared with a tree-based approach, CCLEARNER efficiently detected clones with high precision.

By investigating different parameter settings in DNN, we explored how CCLEARNER's effectiveness varies with the number of hidden layers and the number of iterations. By experimenting with different feature sets, we observed that the similar usage of reserved words, markers, type identifiers, and method identifiers were always good indicators of code clones. However, exactly matching the other kinds of terms to predict clones can cause false positives and/or false negatives, because clones are more likely to use divergent operators, literals, qualified names, and variable identifiers. We tried to use an n-gram algorithm [46] (i.e., bigram) to fuzzily match *similar but different* terms, but this slowed CCLEARNER down significantly. In the future, we will investigate more advanced ways to flexibly match similar terms without incurring too much runtime overhead. We plan to experiment with other machine learning techniques to explore whether deep learning is the best technique to train a classifier for clone detection. We also plan to investigate other ways to detect clones with deep learning, such as relying on DNN to automatically extract features from known code clones.

ACKNOWLEDGMENT

We thank anonymous reviewers for their thorough comments on our earlier version of the paper. This work was partially supported by NSF Grant No. CCF-1565827.

REFERENCES

[1] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *TSE*, pp. 654–670, 2002.

[2] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: scalable and accurate tree-based detection of code clones," in *ICSE*, 2007, pp. 96–105. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.30

[3] C. K. Roy and J. R. Cordy, "NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, vol. 0. Los Alamitos, CA, USA: IEEE, Jun. 2008, pp. 172–181. [Online]. Available: http://dx.doi.org/10.1109/icpc.2008.41

[4] N. Göde and R. Koschke, "Incremental clone detection," in *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, 2009.

[5] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big code," *CoRR*, vol. abs/1512.06448, 2015.

[6] T. Apiwattanapong, A. Orso, and M. J. Harrold, "A differencing algorithm for object-oriented programs," in *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, 2004.

[7] 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 *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014.

[8] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of the Second Working Conference on Reverse Engineering*, ser. WCRE '95. Washington, DC, USA: IEEE Computer Society, 1995.

[9] G. Antoniol, U. Villano, E. Merlo, and M. D. Penta, "Analyzing cloning evolution in the linux kernel," *Information and Software Technology*, 2002.

[10] R. Koschke and S. Bazrafshan, "Software-clone rates in open-source programs written in c or c++," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 3, March 2016, pp. 1–7.

[11] 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, Sept 2007.

[12] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016.

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[14] "ANTLR," http://www.antlr.org/.

[15] "Use JDT ASTParser to Parse Single .java files," http://www.programcreek.com/2011/11/use-jdt-astparser-to-parse-java-file/.

[16] "Introduction to deep neural networks," https://deeplearning4j.org/neuralnet-overview.

[17] C. M. Bishop, *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., 1995.

[18] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, no. 4, pp. 193–202, 1980. [Online]. Available: http://dx.doi.org/10.1007/BF00344251

[19] "ASTVisitor," http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FASTVisitor.html.

[20] "Deeplearning4j," http://deeplearning4j.org/, accessed: 2016-06-26.

[21] *Data Mining Techniques: For Marketing, Sales, and Customer Relationship Management*. Wiley Publishing, 2011.

[22] "BigCloneBench," https://github.com/clonebench/BigCloneBench.

[23] "PostgreSQL: The world's most advanced open source database," https://www.postgresql.org.

[24] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ser. ICSME '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 131–140. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2015.7332459

[25] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.

[26] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *SCHOOL OF COMPUTING TR 2007-541, QUEEN'S UNIVERSITY*, 2007.

[27] B. S. Baker, "A program for identifying duplicated code," *Computing Science and Statistics*, 1992.

[28] J. H. Johnson, "Identifying redundancy in source code using fingerprints," in *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, 1993.

[29] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, 2006.

[30] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the International Conference on Software Maintenance*, 1998.

[31] F. E. Allen, "Control flow analysis," in *Proceedings of a Symposium on Compiler Optimization*, 1970.

[32] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, 1987.

[33] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proceedings of the 8th International Symposium on Static Analysis*, 2001.

[34] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, 2001.

[35] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: detection of software plagiarism by program dependence graph analysis," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2006.

[36] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching for clone and concept detection," in *Reverse Engineering*. Kluwer Academic Publishers, 1996.

[37] J. Mayrand, C. Leblanc, and E. M. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *International Conference on Software Maintenance*, 1996.

[38] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for better configurations: A rigorous approach to clone evaluation," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 455–465. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491420

[39] C. Ragkhitwetsagul, M. Paixao, M. Adham, S. Busari, J. Krinke, and J. H. Drake, *Searching for Configurations in Clone Evaluation – A Replication Study*. Cham: Springer International Publishing, 2016, pp. 250–256. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-47106-8

[40] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports (n)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[41] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[42] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[43] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, 2006.

[44] T. Mikolov, M. Karafiát, L. Burget, J. Cernocký, and S. Khudanpur, "Recurrent neural network based language model," in *INTERSPEECH*, 2010.

[45] C. Goller and A. Kuchler, "Learning task-dependent distributed representations by backpropagation through structure," in *IEEE International Conference on Neural Networks*, 1996.

[46] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig, "Syntactic clustering of the web," in *Selected Papers from the Sixth International Conference on World Wide Web*, 1997.

Running CCLEARNER with BigCloneBench

I. SYSTEM REQUIREMENTS

CCLEARNER could be deployed and executed on Ubuntu at present. We tested the tool in both our lab server and virtual machine. There is no special requirement except for Java 8. The VirtualBox image with CCLEARNER installed is available at https://goo.gl/k6rjDn. Below are our system configurations for the VirtualBox image:

- OS: Ubuntu 14.04 LTS (64-bit),
- Java Version: 1.8.0_131,
- CPU: Intel Core i7-4980HQ @2.80GHz $\times$ 2,
- Harddisk: 60GB, and
- Memory: 4GB.

II. BIGCLONEBENCH PREPARATION

We leveraged the BigCloneBench dataset to train and test the classifier in CCLEARNER. The dataset consists of a Java codebase, and a database that contains information of true and false clone pairs in the codebase. Both tar.gz files (codebase and database) can be downloaded from https://github.com/clonebench/BigCloneBench. After extracting the above two files in the *BigCloneBench* folder, we utilized PostgreSQL (https://www.postgresql.org/) to load the database file. The SQL file contains two hidden roles: **postgresql** and **bigclonebench**, which need to be manually created to ensure a successful SQL file dumping. There are 9 tables in the database and CCLEARNER uses 3 of them: *clones*, *tool*, and *tools_clones*. The *clones* table stores the true clone information; *tool* registers clone detection tools; and *tools_clones* records the clone detection results by each registered tool. CCLEARNER must be registered in the *tool* table by setting the **ID** value as **1**. We chose to use pgAdmin (https://www.pgadmin.org/) as the PostgreSQL client.

III. CCLEARNER DOWNLOAD AND CUSTOMIZATION

CCLEARNER can be downloaded from https://github.com/liuqingli/CCLearner. Each folder with the prefix *CCLearner* contains the source code for a particular stage. The dependencies and executable jar files are in the *Run* folder. The *Recall_Query* folder contains predefined recall queries. The *CCLearner.conf* file is used to specify or tune parameters, including file paths, database settings, training models and testing folders. Figure 1 shows a fragment of the configuration file.

IV. EXECUTION

To evaluate CCLEARNER, we strongly recommend to execute the tool following the instructions on the project website. Run the three jar files (i.e., *CCLearner_Feature.jar*, *CCLearner_Train.jar*, *CCLearner_Test.jar*) in sequence to extract features, train the classifier, and detect clones. Figure 2 shows the screenshot of running *CCLearner_Feature.jar* to extract features, while Figure 3 presents the screenshot for executing *CCLearner_Test.jar* to detect clones.

```
#################################################################
# Feature Configuration
#################################################################
# The number of features should be 8 or 7
# If the number of features is 8, feature name should be null
# If the number of features is 7, select one feature that will be removed
feature.num=8
# Name could be "reservedword", "type", "literal", "variable", "functionname", "
qualifiedname", "operator" and "marker"
feature.name=null
feature.minline=6
```

Fig. 1: CCLearner.conf

```
cclearner@cclearner-VirtualBox:~/Desktop/CCLearner/Run$ java -jar CCLearner_Feature.jar
START!!
Extracting True/False Clone Pairs -T1    : 13750/13750
Extracting True/False Clone Pairs -T2    : 3104/3104
Extracting True/False Clone Pairs -VST3  : 1207/1207
Extracting True/False Clone Pairs -ST3   : 4602/4602
Merging Training Files...
Deleting Old Files...
COMPLETE!!
Time Cost:      119057ms
```

Fig. 2: Execute CCLearner_Feature.jar

```
16:48:28.275 [main] DEBUG o.d.optimize.solvers.BaseOptimizer - Objective functio
n automatically set to minimize. Set stepFunction in neural net configuration to
 change default settings.
---------------------------------------------------
Get files from directory 11...
Finish!
---------------------------------------------------
Get methods from files...
# of methods: 249
Finish!
---------------------------------------------------
Calculating similarity and writing to file...
Method processing: 0/249
Finish!
---------------------------------------------------
Checking clone pairs...
Finish!
```

Fig. 3: Execute CCLearner_Test.jar

V. EVALUATION

To evaluate CCLEARNER's clone detection effectiveness, truncate the *tools_clones* table first, and then import CCLEARNER's clone detection results—a CSV file—into the table. Figure 4 shows the data import screenshot. Open the *Recall_Query* folder, and execute the predefined recall queries with pgAdmin to calculate the recall rates.

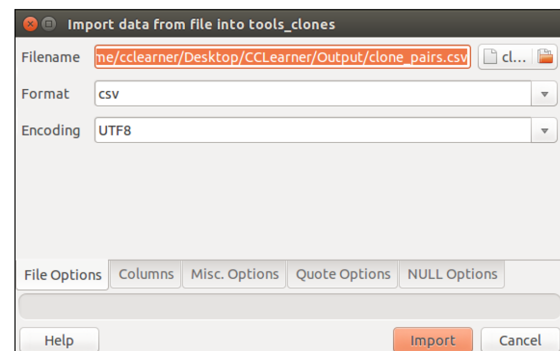

Fig. 4: Import a CSV File to the *tools_clones* table