CCLearner: Clone Detection via Deep Learning

Liuqing Li, He Feng, Na Meng, Barbara Ryder

Abstract To facilitate clone maintenance, various automated tools were proposed to detect code clones by identifying similar token sequences or similar program syntactic structures in source code. They achieved different trade-offs between precision and recall. Inspired by prior work, we developed a new approach CCLEARNER, *a solely token-based clone detection approach using deep learning*. Given known clones pairs and non-clone pairs, CCLEARNER extracts features from each code pair, and leverages the features to train a classifier. The classifier is then used to compare methods pair-by-pair in a given codebase to detect clones. We evaluated CCLEARNER by reusing an existing benchmark of real clone code—BigCloneBench. We split the benchmark such that some data was used for classifier training, and some data was used for testing. With the testing data, we evaluated CCLEARNER's effectiveness of clone detection, and also assessed three existing popular clone detection tools: SourcererCC, NiCad, and Deckard. CCLEARNER outperformed existing tools by achieving a better trade-off between precision and recall. To further investigate whether other machine learning algorithms can perform comparatively as deep learning, we replaced deep learning with five alternative machine learning algorithms in CCLEARNER, and observed that CCLEARNER worked best when using deep learning.

Liuqing Li
Virginia Tech, Blacksburg, VA 24061, e-mail: liuqing@vt.edu

He Feng
Virginia Tech, Blacksburg, VA 24061, e-mail: fenghe@vt.edu

Na Meng
Virginia Tech, Blacksburg, VA 24061, e-mail: nm8247@vt.edu

Barbara Ryder
Virginia Tech (Emeritus), Blacksburg, VA 24061, e-mail: ryder@cs.vt.edu

1 Introduction

In software development, developers copy and paste code to quickly reuse already-implemented functionalities in multiple program contexts. However, the produced code clones may be challenging to track and maintain. To overcome the challenge, researchers built various automated clone detection tools [1, 2, 3, 4, 5, 6]. For instance, SoucererCC indexes code blocks with the least frequent tokens they use, in order to quickly retrieve potential clones of a given code block [5]. NiCad leverages TXL [7] to parse source code, and to convert the parsed syntax trees to a user-specified normalized code representation [3]. NiCad then detects clones by comparing the token sequences of normalized representations of different code. Both SourcererCC and NiCad mostly identify Type-1 (T1) and Type-2 (T2) clones. Deckard parses syntax trees from code, characterizes subtrees with numerical vectors, and detects similar code by comparing numerical vectors [2]. Different from SourcererCC and NiCard, Deckard detects more Type-3 (T3) clones [5].

Inspired by prior work, we designed and implemented CCLearner [8][1], a novel deep learning-based approach to detect clones solely based on tokens. Our insight is that ***tokens (e.g., reserved words, type identifiers, method identifiers, and variable identifiers) provide good indicators of program implementation***. If two code blocks use the same tokens in identical or similar ways, the blocks are likely to be clones and may realize identical or similar features. Furthermore, by treating the clone detection problem analogous to a classification problem that decides whether two blocks are clones or not, we can leverage machine learning (including deep learning) to identify clones. In comparison with former approaches whose clone detection algorithms were manually designed, CCLearner exploits deep learning to train a classifier based on known clones and non-clones. With the classifier automatically characterizing any commonality and variation between clone peers, CCLearner detects clones by enumerating all method pairs in a given codebase and determining which pair has the cloning relationship.

In our evaluation, we experimented with CCLearner and three existing clone detection tools: Deckard [2], NiCad [3], and SourcererCC [5]. We constructed the evaluation data set based on BigCloneBench [9]. CCLearner achieved the best trade-off between precision and recall among all tools. To further evaluate CCLearner's effectiveness when it uses machine learning (ML) algorithms other than deep learning, we also experimented with five alternative ML algorithms in addition to deep learning: AdaBoost [10], Decision Tree [11], Naïve Bayes [12], Random Forest [13], and Support Vector Machine (SVM) [14]. We observed that CCLearner's clone detection effectiveness varied a lot between the adopted ML algorithms.

This chapter extends our recent research publication on CCLearner [8]. In the following sections, we will first overview the published work (Sections 2–4), and then introduce our new experiment and observations after the publication (Section 5). Finally, we will discuss the lessons learned from our investigation and share our thoughts on future research directions.

---

[1] Download link: `https://github.com/liuqingli/CCLearner`

2 Background

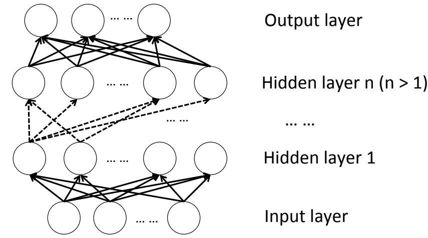

Fig. 1: The DNN architecture

This section first introduces deep neural network (DNN)—the deep learning algorithm used in CCLEARNER, and then clarifies our terminology.

Deep Learning (DL) includes a set of algorithms that can be used to model high-level abstractions in data. Among various DL algorithms, the **deep neural network (DNN)** [15] is a representative algorithm that demonstrated impressive performance in a variety of classification tasks. DNN is an **artificial neural network (ANN)** that has one input layer, one output layer, and two or more hidden layers between the input and output layers (see Fig. 1). Each layer has multiple nodes (i.e., artificial neurons). Every node combines its inputs with the corresponding *weights* or *coefficients* to either amplify or dampen those inputs. During the learning process, all *weights* of nodes in DNN are optimized through backpropagation to minimize the loss between predicted labels and true labels. In this way, each node infers which inputs are more helpful for the overall learning task, and how each input progresses through the network to affect the ultimate outcome, say, an act of classification [16].

In our research, a **clone method pair** or **true clone pair** represents two methods or functions that have similar code. Each method in a true clone pair is denoted as a **clone peer** of the other. Similarly, we define **non-clone method pair** or **false clone pair** to represent any two methods that have very different code from each other. Each method in the false clone pair is called a **non-clone peer** of the other method.

3 Approach

As shown in Fig. 2, CCLEARNER consists of two phases: training (Section 3.2) and testing (Section 3.3). The Feature Extraction procedure (Section 3.1), performed in both phases, extracts eight features from token sequences. In the training phase, CCLEARNER takes in both clone and non-clone method pairs to train a deep learning-based classifier. In the testing phase, given any codebase, CCLEARNER uses the trained classifier to detect clones.

3.1 Feature Extraction

CCLEARNER extracts features that characterize the clone (or non-clone) relationship of any method pair $(method_A, method_B)$. For each method, CCLEARNER first tokenizes the code to identify all tokens, and uses a *token-frequency list* to record the

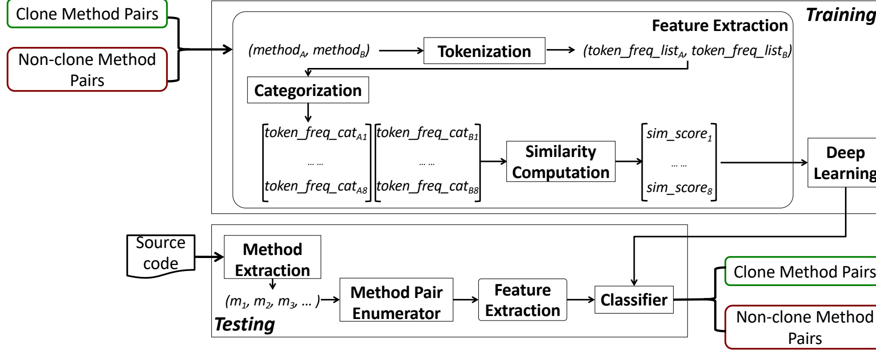

Fig. 2: The overview of CCLearner

occurrence count of each token. In Fig. 2, $token_freq_list_A$ and $token_freq_list_B$ separately represent such token information of $method_A$ and $method_B$. We believe that different kinds of tokens provide distinct signals to indicate code (dis)similarity, so we classified tokens into eight categories and CCLearner splits each method's token-frequency list into eight sublists accordingly. Next, CCLearner computes a similarity score for each pair of token sublists between $method_A$ and $method_B$. The resulting eight similarity scores are then used as features to represent the relationship between methods.

Token Categorization and Extraction. *Different types of tokens may have different capabilities to characterize clones*. For instance, clone peers are more likely to share reserved words (e.g., "`for`" and "`if`") rather than operators (e.g., "`+`" and "`&`"), because they usually have identical program structures but may use slightly different arithmetic or logic operations. Therefore, we classified tokens into eight categories based on their syntactic or semantic meanings. Table 1 presents all token categories and the related exemplar token-frequency sublists.

Table 1: Token categories and related token-frequency sublists

Index	Category name	An exemplar token-frequency sublist
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>, ...

To create token-frequency (sub)lists based on source code, CCLearner uses both the ANTLR lexer [17] and Eclipse ASTParser [18]. Given a code block, the ANTLR lexer extracts all tokens in sequence. As the token sets of reserved words, operators, and markers are well defined, CCLearner recognizes C1–C3 tokens

purely based on the lexer's outputs. The token sets of literals, type identifiers, method identifiers, qualifier names, and variable identifiers vary with codebases, so the ANTLR lexer cannot identify C4–C8 tokens precisely. To overcome the lexer's limitation, we used Eclipse ASTParser to generate an Abstract Syntax Tree (AST) for each method, and implemented several ASTVisitors to traverse trees and to retrieve tokens contained by certain types of AST nodes. For instance, one of the ASTVisitors extracts method identifiers (C6 tokens) by locating and processing all method-relevant AST nodes, including MethodDeclaration and MethodInvocation. Notice that ASTParser complements instead of replacing the ANTLR lexer, because the parser is unable to reveal all tokens that the lexer detects (e.g., reserved words).

Similarity Computation. When two methods are characterized with vectors of token-frequency sublists, we rely on *vector-wise similarities to capture the similarity of method bodies*. Intuitively, the more similar vectors there are between two methods, the more likely those methods are clones to each other. Specifically for each token category $C_i (1 \leq i \leq 8)$, CCLearner computes a similarity score sim_score_i between methods' corresponding token-frequency sublists L_{Ai} and L_{Bi} as below:

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

Here x is a token contained by L_{Ai} or L_{Bi}, $freq(L_{Ai}, x)$ represents the occurrence count of x in $method_A$, and $freq(L_{Bi}, x)$ denotes x's frequency in $method_B$. The computed similarity score varies within [0, 1]. In general, the more tokens shared between lists and the less frequency difference there is for each token, the higher the similarity score becomes. In particular, when the token-frequency sublists of a certain category share no token in common, we set the corresponding similarity score to 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 during experiment. This may be because when no token is commonly shared between sublists, the frequency distributions may be similar or not; 0.5 does not suggest any bias towards either similarity or dissimilarity.

3.2 Training

We need both positive and negative examples to train a classifier for binary classification. CCLearner takes feature vectors extracted from clone method pairs as positive examples and feature vectors derived from non-clone pairs as negative examples. Each data point for training is represented as $< similarity_vector, label >$, where $similarity_vector$ is an eight-dimensional vector of similarity scores, and $label$ is either 1 ("$CLONE$") or 0 ("NON_CLONE"). To avoid any confusion caused by small clone methods, we refined our training data with methods that contained at least six lines of code. 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. We used DeepLearning4j [19] to train a DNN classifier. The input layer contains eight nodes, with each node taking one feature value in $similarity_vector$. The output layer predicts whether a method pair is "$CLONE$" or "NON_CLONE". CCLearner configures DNN to include 2 hidden layers and to run 300 iterations for training, as CCLearner worked best with these parameter settings in our experiment. Each hidden layer is configured to have 10 nodes, as suggested by literature [20].

3.3 Testing

Given a codebase, CCLearner first detects methods from source files with Eclipse ASTParser, and then enumerates all possible method pairs. CCLearner feeds each enumerated method pair to the trained classifier for clone detection. Theoretically, when n methods are extracted from a codebase, the clone detection algorithm complexity should be $O(n^2)$. To reduce the comparison run-time overhead, we developed two heuristics to filter out some unnecessary comparison. One filter was applied to examine 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, so we simply conclude that they compose a non-clone method pair and skip any further processing. Another filter removes any candidate method having less than six LOC for two reasons. First, small methods may contain so few tokens that CCLearner cannot effectively perform clone detection. Second, the six-line minimum is common in clone detection benchmarks mentioned in prior research [21, 5]. In CCLearner, the output layer has two nodes to separately predict the likelihood of clones and non-clones: l_c and l_{nc}, where $l_c + l_{nc} = 1$. We set $l_c \geq 0.98$ to detect clones as precisely as possible without producing many false alarms.

4 Empirical Comparison with Existing Tools

To assess how CCLearner compares with existing tools, we created training and testing sets based on a large-scale clone benchmark (Section 4.1). We also defined metrics to measure the effectiveness of automatic clone detection (Section 4.2). By applying CCLearner and three alternative tools on the datasets, we quantitatively evaluated different tools' capabilities of clone detection (Section 4.3).

4.1 Dataset Generation

BigCloneBench [9] is a large-scale code clone benchmark. It consists of two parts: a codebase and a database. Table 2 summarizes the clone data contained by our

downloaded reduced version of BigCloneBench [22]. As shown in the table, BigCloneBench's codebase has 10 source folders. Each folder has multiple Java files, and each file contains various Java methods. Every method independently implements one functionality (e.g., sorting). BigCloneBench's database stores clone information related to the codebase. It records over 6 million recognized true clone pairs and 260 thousand false clone pairs in the codebase. For each method pair, the database records their code locations and clone type information. Specifically, **VST3**, **ST3**, **MT3**, **WT3/4** respectively represent "Very Strong T3", "Strong T3", "Medium T3", "Weak T3 or T4" clones.

Table 2: Data in the downloaded BigCloneBench

Folder Id.	# of 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 3: Datasets for training and testing

Dataset	# 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

Compared with other folders, **Folder #4** has the largest number of both true and false clone pairs. Therefore, we leveraged the data in this folder for training and the data in all other folders for testing. As MT3 and WT3/4 clones can 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. We also removed all methods that have five or fewer LOC to reduce data noise. As **Folder #4** has a lot more false clone pairs than true clone pairs, we randomly sampled a subset of false clone pairs to achieve a count balance between the positive examples and negative examples used in training. Table 3 presents the resulting datasets we created based on BigCloneBench.

4.2 Evaluation Metrics

We defined three metrics to evaluate the effectiveness of automatic clone detection:

Recall measures a tool's ability to retrieve true clones; it is the fraction or percentage of known true clone pairs that are detected by any clone detection approach. Taking the labeled clones in BigCloneBench as known true clones, we could automatically evaluate an approach's recall for individual clone types. Since many tools cannot effectively retrieve MT3 and WT3/4 clones, as with prior work [5], we evaluated the overall recall for T1, T2, VST3, and ST3 clones as below:

$$R_{T1-ST3} = \frac{\text{\# of true clone pairs (of T1-ST3)}}{\text{Total \# of known true clone (of T1-ST3)}}. \quad (2)$$

Precision measures a tool's ability to correctly report true clones; it is the fraction of true positives among all clone pairs reported by a tool. The labeled clones in BigCloneBench cannot be used to automatically compute precision, because based on our experience, the labeled dataset misses some true clones actually existing in the codebase. Instead, we need to manually inspect all clones reported by any tool to decide the precision rates. When a clone detection tool reports thousands of clone pairs, we cannot afford the manual effort to inspect every pair. Therefore, in our evaluation, we manually examined a sample set of clones reported by each tool. To ensure that our sampled data is representative, we chose 385 reported clones among all clone types for each approach. The number 385 is a statistically significant sample size with a 95% confidence level and ±5% confidence interval, when the population size is larger than 200,000. With the manual inspection of 385 sampled clone pairs, we estimated precision as below:

$$P_{estimated} = \frac{\text{\# of true clone pairs}}{\text{385 detected clone pair samples}}. \quad (3)$$

C score combines precision and recall to measure the overall accuracy of clone detection. It is calculated as the harmonic mean of R_{T1-ST3} and $P_{estimated}$:

$$C = \frac{2 * P_{estimated} * R_{T1-ST3}}{P_{estimated} + R_{T1-ST3}}. \quad (4)$$

4.3 Effectiveness Comparison of Clone Detection Approaches

To evaluate CCLearner's capability of clone detection, we compared it with three popular tools: SourcererCC [5], NiCad [3], and Deckard [2]. We applied all three existing tools to CCLearner's testing data with their default tool configurations.

Recall. As shown in Table 4, CCLearner achieved the highest recall among all tools when detecting T1-ST3 clones; it was unable to detect as many MT3 and WT3/4 clones as Deckard did. Specifically, CCLearner identified all T1 clones,

Table 4: Recall comparison among tools (%)

Tool	T1	T2	VST3	ST3	MT3	WT3/4	Average (R_{T1-ST3})
CCLearner	100	98	98	89	28	1	93
SourcererCC	100	97	92	67	5	0	80
NiCad	100	85	98	77	0	0	85
Deckard	96	82	78	78	69	53	83

98% of T2 and ST3 clones, and 89% of ST3 clones. From T1 to WT3/4, as clone peers become less similar, CCLEARNER's recall decreased. The same trend was also observed for other tools, which could be explained by the increased difficulty of clone detection as clone peers become more dissimilar to each other. CCLEARNER was unable to achieve 100% recall for all clone types, mainly because it relies on the exactly same terms used in method pairs to compute similarity vectors. When two clone methods share few identifiers and contain significantly divergent program structures, CCLEARNER cannot reveal the clone relationship. In the future, we plan to devise supplementary techniques for these specialist clones.

Table 5: Comparison among tools for the sampled precision, and the number of reported and true clone pairs

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

Precision. Table 5 shows the $P_{estimated}$, the number of reported clone pairs, and the number of estimated *true* clone pairs for all tools. Notice that the number of *true* clones was calculated as the product of $P_{estimated}$ and total number of reported clones. Compared with SourcererCC and NiCad, CCLEARNER reported more true clone pairs. Deckard had a lower $P_{estimated}$ of 71% but reported more clones than any other tool. These numbers indicate that Deckard retrieved more true clones and produced more false alarms (wrongly reported clones). This may be because Deckard flexibly matches code snippets by tolerating more differences in the token usage and program structures.

Table 6: Tool comparison for C scores and runtime costs

Tool	*C* (%)	Runtime cost
CCLearner	93	47min
SourcererCC	88	13min
NiCad	76	34min
Deckard	77	4h 24min

C Score and Time Cost. Table 6 lists different tools' C scores and execution time. CCLEARNER obtained the highest *C* score, which implies that CCLEARNER detected

clones more accurately by achieving both high estimated precision and high T1-ST3 recall. Regarding tools' runtime overhead, SoucererCC ran the fastest (spending 13 minutes). NiCad was slower than SourcererCC (spending 34 minutes). Deckard worked the most slowly (spending 4 hours and 24 minutes), because it used an expensive tree matching algorithm. Our observations on the above results align with the findings in prior work [5]. CCLearner took 47 minutes to detect clones. Similar to SourcererCC and NiCad, CCLearner worked faster than Deckard since it did not reason about program structures. However, CCLearner was slower than NiCad and SourcererCC, because it calculated $similarity_vector$ and compared methods pair-by-pair to find clones. In addition, CCLearner spent another five minutes on training, which could be considered as one-time cost and ignored. Due to the pair-wise comparison mechanism, CCLearner's clone detection is an embarrassingly parallel task [23], indicating that we can easily parallelize the task to further reduce CCLearner's time cost in the future.

5 CCLearner Sensitivity to Machine Learning Algorithm Used

By default, CCLearner uses DNN to train a classifier for clone detection. To explore how well CCLearner works when it adopts different machine learning (ML) algorithms, after the published work [8], we also experimented with five alternative ML algorithms: AdaBoost [10], Decision Tree [11], Naïve Bayes [12], Random Forest [13], and Support Vector Machine (SVM) [14]. Specifically for algorithm implementation, we used Weka [24] because it is a software library implementing a collection of ML algorithms (including all five algorithms mentioned above). By replacing DNN with each alternative, we trained five distinct learners and thus obtained five tool variants: CCLearner$_a$, CCLearner$_d$, CCLearner$_n$, CCLearner$_r$, and CCLearner$_s$. For fair tool comparison, when exploiting each alternative ML algorithm, we reused the training and testing data shown in Table 3 to train a classifier and to evaluate the resulting clone detection effectiveness.

Table 7: The effectiveness of CCLearner and its variants (%)

Tool	ML Algorithm	Recall Per Type						R_{T1-ST3}	$P_{estimated}$	C
		T1	T2	VST3	ST3	MT3	WT3/4			
CCLearner	DNN	100	98	98	89	28	1	93	93	93
CCLearner $_a$	AdaBoost	100	98	98	95	58	4	97	63	76
CCLearner $_d$	Decision Tree	100	98	98	96	61	4	97	59	74
CCLearner $_n$	Naïve Bayes	100	98	98	95	59	4	97	70	81
CCLearner $_r$	Random Forest	100	98	98	96	62	5	97	56	71
CCLearner $_s$	SVM	100	98	98	96	62	5	97	46	63

Effectiveness Comparison between CCLearner and Its Variants. Table 7 presents the evaluation results by CCLearner and its five variants. For instance, row **CCLearner** corresponds to the results by the default implementation using DNN; and row **CCLearner**$_a$ shows results by the AdaBoost-based implementation. We

observed an interesting phenomenon in Table 7. *Compared with CCLEARNER, all variant approaches obtained higher recall rates, lower precision rates, and lower C scores*. Specifically, all variants' overall recall rates are surprisingly identical: 97%. This number is higher than CCLEARNER's recall: 93%. All variants retrieved MT3 and WT3/4 clones much more effectively than CCLEARNER. For instance, both Random Forest and SVM led to the same highest MT3 and WT3/4 recall rates (*i.e.*, 62% and 5%). Meanwhile, the variants' precision rates are much lower than CCLEARNER's, ranging from 46% to 70%. Overall, CCLEARNER acquired the highest C score—93%, while CCLEARNER$_n$ obtained the second highest C score: 81%. SVM produced the lowest C score: 63%. Although different ML algorithms were applied to the same data, they presented distinct trade-offs between precision and recall. The variants often found more true clones, but those true clones were always mixed in with a larger number of false clones than would be reported by CCLEARNER.

(a) Folder #2, 1937566.java

```
private JSONObject executeHttpGet(
      String uri) throws Exception{
   HttpGet req=new HttpGet(uri);
   HttpClient client=new
         DefaultHttpClient();
   HttpResponse resLogin=client.execute(
         req);
   BufferedReader r=new BufferedReader(
         new InputStreamReader(resLogin.
         getEntity().getContent()));
   StringBuilder sb=new StringBuilder();
   String s=null;
   while((s=r.readLine())!=null) {
     sb.append(s);
   }
   return new JSONObject(sb.toString());
}
```

(b) Folder #2, 2571726.java

```
public static String getStringResponse
      (String urlString) throws
      Exception{
   URL url=new URL(urlString);
   BufferedReader in=new BufferedReader
         (new InputStreamReader(url.
         openStream()));
   String inputLine;
   StringBuilder buffer=new
         StringBuilder();
   while((inputLine=in.readLine())!=
         null) {
     buffer.append(inputLine);
   }
   in.close();
   return buffer.toString();
}
```

Fig. 3: An MT3 clone pair detected by all variants but missed by CCLEARNER

Table 8: Time cost comparison

Tool	Time Cost
CCLEARNER	47min
CCLEARNER$_a$	49min
CCLEARNER$_d$	49min
CCLEARNER$_n$	50min
CCLEARNER$_r$	49min
CCLEARNER$_s$	52min

A Case Study. To understand how CCLEARNER's variants detect clones differently from CCLEARNER, we sampled 10 reported clone pairs for manual inspection. Among the 10 pairs, 5 pairs were identified by CCLEARNER but missed by some of its variants, while the other 5 pairs were revealed by all variants but missed by CCLEARNER. We observed that each of the variants achieved higher recall and lower precision because they tolerated more differences between clones. Figure 3 presents a clone pair detected by all variants but missed by CCLEARNER. The clone peers invoke different methods (*e.g.*, `HttpGet()` and `URL()`) and use different types (*e.g.*, `HttpClient` and `InputStreamReader`). The tool variants reported this clone pair by matching code more flexibly than CCLEARNER. They achieved different trade-offs between preci-

sion and recall, and all outperformed CCLearner in terms of recall at the cost of sacrificing precision.

Effectiveness Comparison between Learning-Based and Non-Learning-Based Approaches. We compared the effectiveness of variant approaches with the results of non-learning-based tools shown in Tables 4-6. We found that the learning-based approaches obtained higher recalls but lower precisions. Specifically, all CCLearner's variants achieved the same R_{T1-ST3}: 97%. This number is much higher than the recall rates of non-learning-based approaches (i.e., SourcereCC, NiCad, and Deckard), which were 80%–85%. All variants' MT3 and WT3/4 recall rates are lower than Deckard's, but higher than those rates of SoucererCC and NiCad. On the other hand, the learning-based variants obtained relatively lower precision rates (46%–70%) than non-learning-based approaches (68%–98%). Although the Naïve Bayes-based approach (CCLearner$_n$) achieved the highest precision rate (70%) among all variants, the rate is only comparable to that of NiCad (68%) or Deckard's (71%). Overall, CCLearner$_n$ worked better than the other variants, but its C score (81%) is much lower than SourcereCC's 88%—the highest C score obtained by the explored non-learning-based approaches.

Table 8 presents the time cost comparison between CCLearner and its variants. All variants have similar time costs to CCLearner, with slightly higher runtime overhead. According to Table 6, all these variants are slower than SourcererCC and NiCad, but faster than Deckard.

6 Conclusion

We designed and implemented a deep learning-based clone detection approach—CCLearner. Different from traditional clone detection tools, CCLearner does not contain manually defined rules or algorithms to specially characterize clones. Instead, it computes token-level similarity vectors between given code blocks, and relies on DNN to characterize the similarity vectors for both clone pairs and non-clone pairs. More learning-based clone detection tools have been recently proposed [25, 26, 27, 28, 29]. These tools process source code to extract tokens, ASTs, control flows and/or data flows, and to create vectorized program representations accordingly; they also adopt more complex neural networks (e.g., recurrent neural network [30], recursive neural network [31], and convolutional neural networks [32]) to take in the vector representations and to better learn the characteristics of clone pairs. All these approaches demonstrate the nice fusion of static program analysis and deep learning. They also evidence that to better interpret the syntax or semantics of programs, we need to improve both program representations and machine learning architectures.

We foresee that more and more learning-based approaches will be proposed in the future to detect clones, analyze code, locate bugs, and repair code. As the future research directions, we plan to conduct empirical comparison between similar tools and understand which learning-based approach design is superior to others. Additionally, we are also curious about the limitation of deep learning (DL)-based

approaches. It seems that DL is good at performing certain tasks and perhaps bad at doing other things. Although it is still unclear what is the domain where DL does not quite fit, we will explore more usage of DL in Software Engineering research to better characterize its application scope.

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. R. Cordy, "The txl source transformation language," *Sci. Comput. Program.*, vol. 61, no. 3, pp. 190–210, 2006.
8. L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 249–260.
9. 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.
10. B. Kégl, "The return of adaboost.mh: multi-class hamming trees," *CoRR*, vol. abs/1312.6086, 2013. [Online]. Available: http://arxiv.org/abs/1312.6086
11. J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, 1986.
12. S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Pearson Education, 2003.
13. T. K. Ho, "Random decision forests," in *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*, 1995.
14. C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, 1995.
15. 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
16. "Introduction to deep neural networks," https://deeplearning4j.org/neuralnet-overview, last accessed: 2017-02-15.
17. "ANTLR," http://www.antlr.org/, last accessed: 2020-12-03.
18. "Use JDT ASTParser to Parse Single .java files," http://www.programcreek.com/2011/11/use-jdt-astparser-to-parse-java-file/, last accessed: 2020-12-03.
19. "Deeplearning4j," http://deeplearning4j.org/, last accessed: 2020-12-03.
20. *Data Mining Techniques: For Marketing, Sales, and Customer Relationship Management*. Wiley Publishing, 2011.

21. 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.
22. "BigCloneBench," https://github.com/clonebench/BigCloneBench#era-updated, last accessed: 2020-12-03.
23. M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.
24. "Weka 3 - data mining with open source machine learning software in java," https://www.cs.waikato.ac.nz/ml/weka/, last accessed: 2020-12-03.
25. 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.
26. 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*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 141–151. [Online]. Available: https://doi.org/10.1145/3236024.3236068
27. L. Büch and A. Andrzejak, "Learning-based recursive aggregation of abstract syntax trees for code clone detection," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 95–104.
28. 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)*, 2019, pp. 70–80.
29. Z. Gao, V. Jayasundara, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 394–397.
30. L. C. Jain and L. R. Medsker, *Recurrent Neural Networks: Design and Applications*, 1st ed. USA: CRC Press, Inc., 1999.
31. C. Goller and A. Kuchler, "Learning task-dependent distributed representations by backpropagation through structure," in *Proceedings of International Conference on Neural Networks (ICNN'96)*, vol. 1, 1996, pp. 347–352 vol.1.
32. K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *CoRR*, vol. abs/1511.08458, 2015. [Online]. Available: http://arxiv.org/abs/1511.08458