IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 14, 2021, accepted April 16, 2021, date of publication May 11, 2021, date of current version June 18, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3079156

An Effective Semantic Code Clone Detection Framework Using Pairwise Feature Fusion

ABDULLAH SHENEAMER[1], SWARUP ROY[2], AND JUGAL KALITA[3]
[1]Faculty of Computer Science and Information Technology, Jazan University, Jazan 45142, Saudi Arabia
[2]Department of Computer Applications, Sikkim University, Gangtok 737102, India
[3]College of Engineering and Applied Science, University of Colorado at Colorado Springs, Colorado Springs, CO 80918, USA

Corresponding authors: Abdullah Sheneamer (asheneamer@jazanu.edu.sa) and Swarup Roy (sroy01@cus.ac.in)

This work was supported by the Jazan University, Saudi Arabia.

ABSTRACT Code clone detection is important for effective software maintenance. The task is more challenging when clones are semantically similar (Type-IV) in nature, having no structural resemblance to each other. Most existing methods use sequence similarity and/or graph isomorphism between either Abstract Syntax Trees (AST) or Program Dependency Graphs (PDG) to detect Type-I, II and III clones. However, they are mostly unsuccessful in detecting semantic or Type-IV clones. In this work, we propose a novel detection framework using machine learning for automated detection of all four type of clones. The features extracted from a pair of code blocks are combined for possible detection of a clone with respect to a reference block. We use AST and PDG features of both code blocks to prepare labelled training samples after fusing the two feature vectors using three different alternatives. We use six state-of-the-art classification models including Deep Convolutional Neural Network to assess the prediction performance of our scheme. To access the effectiveness of our framework we use seven datasets and compare its performance with five state-of-the-art clone detectors. We also compare a large number of algorithms for code clone detection. Comparing the performance of a large number of machine learning techniques, ANN and non-ANN, using such features, and establishing that fusing of AST and PDG features gives competitive results using deep learning as well as boosted tree algorithms, we find that boosted tree algorithms like XGBoost are quite competitive in clone detection. Experimental results demonstrate that our approach outperforms existing clone detection methods in terms of prediction accuracy.

INDEX TERMS Machine learning, code clones, semantic clones, AST, PDG, features, deep learning, classification.

I. INTRODUCTION

In the software engineering life cycle, maintenance is the most expensive and time-consuming phase. The task of maintenance is arduous usually because of inherent complexity and poor programming practices. In a large software system, it has been observed that often pairs of segments occurring in different locations are functionally identical or similar. Sloppy or even good programmers find it easy to make minor modifications to an existing code segment to serve the current purpose in some other part of a program or a project. Very often programmers find sets of useful statements, called *code blocks*, and copy-paste them as necessary, modifying as per requirement to make the software development process faster. Duplicated code blocks are popularly known as *code clones* (CC). Research has reported that 7%-23% of large software projects are code clones [1], [2]. Many studies show that a software system with frequent occurrence of code clones is difficult to maintain [3]. One of the problems with code cloning occurs when an original code block, which is cloned, contains a bug, causing ripple effects to all cloned blocks distributed all over the program or project. Detecting code clones is an important and challenging task. Automatic detection of clones not only improves the software maintenance task, but also may be useful in identifying software plagiarism [4] and code obfuscation [5], detection of malicious software [6], discovery of context-based inconsistencies [7], and opportunities for code refactoring [8].

Automatic clone detection is an active area of research. A number of efforts to detect clones effectively have been published. Existing clone detection methods commonly use similarity metrics to compare fragments of codes.

The associate editor coordinating the review of this manuscript and approving it for publication was Michael Lyu.

All published methods have difficulty in detecting semantic clones, the most challenging types of clones. Semantic clones are syntactically different, but functionally they produce similar outcomes. Traditional approaches are ineffective because their similarity metrics do not capture semantics well [9], [10]. As a result, performance of the methods becomes fairly low in terms of assessment metrics. Machine learning has been recently used successfully in several approaches for automatic detection of code clones, although the amount of work is limited. Moreover, although there are a few attempts at using machine learning, the efforts have been limited in addressing the issue of semantic clones for detection of code clones.

The contributions of this paper are following.

- We present a simple formal model of the code clone problem and its types to better understand the issues involved.
- We explore a new way of using features from Abstract Syntax Trees (ASTs) and Program Dependency Graphs (PDGs) to detect Java code clones, including semantic clones. We believe that this attempt is the first of its kind to use features from both ASTs and PDGs to detect semantic code clones using machine learning. We use the full path traversal algorithm for extracting AST and PDG features and represent these features as vectors.
- We propose a generalized machine learning framework for clone detection of all four types. Special emphasis is on detecting semantic clones, which is the most challenging type of clones to detect.
- We use state-of-the-art classification models to evaluate the effectiveness of our proposed idea. We also compare the performance of a large number of machine learning techniques, ANN and non-ANN, using such features, and establish that fusing of AST and PDG features gives competitive results using deep learning as well as boosted tree algorithms.

We organize the paper as follows. Section II introduces the code clone detection problem. Prior research in the area is highlighted in Section III. In Section IV, we propose a new machine learning framework for detection of semantic code clones. We evaluate and compare our proposed method and report results in Section V. Finally, we conclude our work in Section VI.

II. DETECTION OF CODE CLONES

Code clone detection may be performed within a single program or project, or across programs or projects. A modular program usually consists of a set of sub-programs or methods. A *method* is a set of executable program statements with precisely defined starting and ending points, performing a cohesive task. In this paper, we term it a *method block*. A method block may be divided into sub-blocks, e.g., loops, conditional statements, etc. In our work, we use the terms method block and *code block* interchangeably.

Definition 1 (Block): A block B is a sequence of statements, $S_i, i = 1, \ldots, M$, comprising of programming language specific executable statements such as loops, logical statements and arithmetic expressions:

$$B =< S_1, \ldots S_M > .$$

Definition 2 (Code Clones): Two code blocks B_i and B_j constitute a code clone pair if they are similar based on some metric:

$$clone(B_i, B_j) = \begin{cases} 1, & \text{if } sim(B_i, B_j) > \theta \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

We measure similarity considering a set of characteristics or features we use to describe a block. We can describe a block simply in terms of the statements contained in it, or in terms of other characteristics extracted from the statements in the code, as we will see later. B_i and B_j are clones, if they score higher than a specific threshold using a pre-specified similarity criterion (*sim*).

The code clone detection problem can be defined as follows.

Definition 3 (Code Clone Detection): Given a pair of blocks B_i and B_j, code clone detection is a boolean mapping function $f : B_i \times B_j \rightarrow N \in [1, 0]$, where f is an implementation of represents the similarity function given in Equation 1.

To detect if a pair of blocks are clones of each other, two kinds of similarities may be considered. Blocks B_i and B_j may be textually similar, or may functionally perform similar tasks or the same task without being textually similar. The first kind of clones is simple in nature, usually resulting from the practice of copying and direct pasting. However, the second type of similarity is difficult to define precisely. Bellon *et al.* [11] identified three type of clones based on textual similarity of the programs.

Definition 4 (Type-I: Exact Clones): Two blocks are the exact clones of each other if they are exactly the same except whitespaces, blanks and comments.

Let B_i and B_j be two blocks. Let $B_i =< S_{i1}, \ldots, S_{iN_i} >$, and $B_j =< S_{j1}, \ldots, S_{jN_j} >$. Let $B_i^t = trim(B_i)$ where $trim(.)$ be a function that removes whitespaces, blanks and comments from the block and its statements. Thus, whitespaces that cover an entire line are removed, as well as whitespaces within statements. B_i and B_j are exact clones of each other if i) $|B_i^t| = |B_j^t|$, i.e., they are both of the same length after trimming, and ii) $\forall k, k = 1, \ldots, |B_i^t|\ S_{ik}^t \equiv S_{jk}^t$ where $\equiv$ means that the two statements are exactly the same, considered as strings. The superscript t means after trimming.

Definition 5 (Type-II: Renamed Clones): Two blocks are the renamed clones of each other if the blocks are similar except for names of variables, identifiers, types, literals, layouts, whitespaces, blanks and comments.

Let B_i^n and B_j^n be two trimmed and normalized blocks: $B_i^n = norm(trim(B_i))$ and $B_j^n = norm(trim(B_j))$ where $norm(.)$ is a literal normalization function. Normalization

replaces all the variables from B_i and B_j with generic variable names, among other operations.

Formally, B_i and B_j are renamed clones if i) $|B_i^t| = |B_j^t|$, i.e., they are both of the same length after trimming and normalizing, and ii) $\forall k, k = 1, \ldots, |B_i^t|\ S_{ik}^n \equiv S_{jk}^n$.

Definition 6 (Type-III: Gapped clones): Two copied blocks are gapped clones if they are similar, but with modifications such as **added or removed statements**, and the use of different identifiers, literals, types, whitespace, layouts and comments. The new flexibility introduced is the addition or removal of statements. Assume we are given two blocks B_i and B_j, and let B_i^n and B_j^n be their trimmed versions, as described earlier. Two gapped sequences can be aligned using various techniques that generate an alignment score (*ascore*) for each alignment [12], [13]. The value of *ascore* is obtained by considering the costs of gaps, and the costs of character mismatches and replacements between the two strings.

We say B_i and B_j are gapped clones of each other if $ascore(B_i^n, B_j^n) > \theta$ for a user-defined threshold θ.

The fourth type of clones is semantic clones. Semantic clones are the most challenging type of clones. Instead of comparing program texts which is relatively easy to do, semantic clones are difficult to identify as they deal with the meaning or purpose of the blocks, without regards to textual similarity. A real life example of semantic clones is a pair of obfuscated blocks or programs [14], where syntax-wise the blocks are by and large different from each other, but the overall meanings of both are the same.

Definition 7 (Type-IV: Semantic clones): Two blocks are semantic clones, if they are semantically similar without being syntactically similar. In other words, two blocks B_i and B_j are semantic clones if

$$semsim(B_i, B_j) = semsim(B_i^n, B_j^n) > \theta, \tag{2}$$

where $semsim(.,.)$ is a semantic similarity function.

The idea of semantic similarity is not easy to grasp because it requires some level of understanding the meanings of programs, whether formal or otherwise. The formal semantics of a program or a block can be described in several ways, the predominant ones being denotational semantics, axiomatic semantics and operational semantics [15], [16]. Denotational semantics composes the meaning of a program or a block by composing it from the meaning (or denotation, a mathematical expression or function) of its components in a bottom-up fashion. Axiomatic semantics defines the meaning of a program or block by first defining the meanings of individual commands by describing their effects on assertions about variables that represent program states, and then writing logical statements with them. Operational or concrete semantics does not attach mathematical meanings to components within a program or block, but describes how the individual steps of a block or program take place in a computer-based system on some abstract machine. No matter which approach is used for describing formal semantics, the meaning of a block or program is obtained from the meanings ascribed to the individual components. To obtain the semantics of a block or a program, it is initially parsed into syntactic or structural components, and for each syntactic component, its corresponding meaning is obtained, and finally the meaning of the block is put together from these components, following appropriate rules. Thus, we could say two blocks B_i and B_j are semantic clones if

$$semsim(B_i^n, B_j^n) = semsim([\![B_i^n]\!], [\![B_j^n]\!]), \tag{3}$$

where $[\![B]\!]$ represents the semantics of a block B, obtained using one of the means for formal semantics. In practice, we should note that the "semantics" of a block may be computed without resorting to formal semantics.

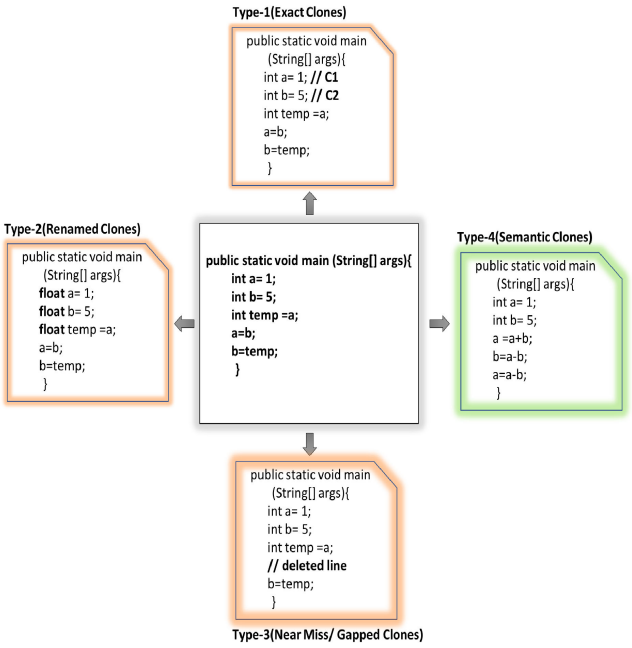


FIGURE 1. Simple example of different types of clones.

Different types of clones are illustrated with the help of a few simple programs in Figure 1. The original code block, in the center of the figure, swaps values of two integer variables using a temporary variable. The Type-I clone is an exact replica of the original code block or program. In case of Type-II, only a few of the literals are changed. The gapped clone block is a replica of the original except that a line has been deleted. The Type-IV clone block (top right) shows another approach to swap two numeric variables without using a third variable. Structurally, the code blocks are dissimilar; however, because the purpose of both code blocks is the same, semantically they are similar. On the other hand, the Type-I through III clone blocks are structurally similar although what they do are different.

III. PRIOR RESEARCH

Several methods have been proposed to detect clones of Type-I, II and III. Interestingly, very few attempts have been made to detect Type-IV or semantic clones. They can be

classified as machine learning and non-machine learning approaches. Non-machine learning methods primarily use syntactical features of a target code pair, and compute a similarity score to declare a piece of code as clone if the score is above a certain threshold.

A. CLASSICAL APPROACHES

CCFinder [17] applied a rule-based transformation on the input source text and compared token-by-token to detect code clones. It used a suffix tree-matching algorithm, and it was not effective in detecting Type-III or IV. NiCad [18] was a text-based hybrid clone detection technique, which could detect up to Type-III clones. NiCad used identification and normalization of potential clones using longest common subsequence matching. Yuan and Guo [4] used a count matrix to detect code clones. The matrix is created by counting the occurrence frequencies of every variable. The technique could detect many hard-to-detect code clones. It constructed and compared two bipartite graphs derived from two code blocks. The method was limited to detecting only Type-I, II, and III clones. Komondoor and Horwitz [19] used for the first time the idea of Program Dependency Graphs (PDG) in clone detection. They used a slicing technique to find isomorphic PDG subgraphs to detect semantic as well as synthetic code clones. Scorpio [20] was another PDG-based approach that used incremental two-way slicing for detecting code clones, limited to only Type-I to III. Sheneamer and Kalita [21] proposed a hybrid clone detection technique that first used a coarse-grained technique to improve precision and then a fine-grained technique to improve recall. It could detect only syntactic clones (Type-I through Type-III). SourcererCC [22] was a token-based syntactic and semantic clone detection method that used an optimized partial index of tokens and filtering heuristics to achieve large-scale detection.

B. MACHINE LEARNING IN CLONE DETECTION

The methods mentioned above use a pairwise similarity measure with respect to certain tree representations of the programs or code blocks. However, for a large program with multiple blocks, it is difficult to match trees pairwise. Two similar program blocks may have similar patterns within them; in other words similar feature signatures. Instead of developing custom algorithms for similarity matching, a machine learning algorithm may be used to learn patterns that differentiate clones and non-clones and also among clones of various types. We present a machine learning framework to automatically detect clones in software, to detect Types-I-III and the most complicated kind of clones, Type-IV clones.

Previously used traditional features are often weak in detecting semantic clones. The novel aspects of our approach are the extraction of features from abstract syntax trees (AST) and program dependency graphs (PDG), representation of a pair of code fragments as a single vector, fusion of features of individual blocks to obtain features of a pair of blocks and the use of classification algorithms. The key benefit of this approach is that our tool can find both syntactic and semantic clones extremely well. Our evaluation indicates that using our new AST and PDG features is a viable methodology, since they improve detecting clones on a very large dataset like IJaDataset2.0. In machine learning terms, a feature is simply a pattern at a certain level–low to high–that a machine learning algorithm extracts in the data. A deep learning method is able to capture features at various levels in the various layers.

Deckard was a tree-based technique [23], which computed characteristic vectors from the AST and clustered these vectors using unsupervised machine learning. Wang *et al.* [24] proposed a syntactic clone detection method using a Bayesian Network framework with features based on developmental history of the code, the actual current text of the code and the destination where the code is pasted. Yang *et al.* [25] used an approach based on generalized suffix trees to detect an initial set of clones. Since a clone detector produces many irrelevant clones, they used an iterative process where users marked up clones as relevant, and used this information with TF-IDF representation to find clones that are highly relevant to specific users. Sæbjornsen *et al.* [26] used a disassembler to recover assembly code; broke up a code block into small regions and represented small regions of normalized code as vectors using features such as op code types, operand types, and n-grams of various kinds; obtained locality sensitive hashes of the small regions and produced vector representations of a code block with a sequence of hashes for the regions; and finally computed l_1-distance between pairs of code blocks to find clones. Cesare *et al.* [27] detected clones of packages, using more than 30 features such as number of files, number of common or similar file names, sizes of packages and package dependencies. They used a variety of machine learning algorithms such as Naive Bayes, Multilayer Perceptrons, Decision Trees and Random Forests.

Li *et al.* [28] extracted tokens from known method-level code clones and non-clones to train a 6-layerd perceptron, and then used the classifier to detect syntactic clones. Shalev and Partush [29] detected similarities between code blocks by breaking them up into smaller regions of a few lines of assembly code; normalizing these code regions; obtaining small sized MD5 hashes of the regions; creating a fixed size hash for a code block; and then training a 4-layered perceptron to detect if two code blocks are clones. Wei and Ming [30] developed an end-to-end approach to detect Type IV clones, using Long Short Term Memory (LSTMs), a special type of RNNs. They trained their network using pairs of code blocks labeled as clones or non-clones to learn to compute hash codes. During testing, code blocks are compared directly using the trained network. Wei and Li [31] used an AST-based modified LSTM architecture that operated on trees to encode code blocks, and then used adversarial training to learn to detect clones. In adversarial training, vectors corresponding to real pairs of examples are perturbed so that learning is more robust. Saini *et al.* [32] used a so-called Siamese twin neural network architecture, whose inputs are obtained by computing features of code blocks. Examples of features

TABLE 1. Brief summary of clone detection methods.

Tools/Alogorithms	Syntactic Clones	Semantic Clones	Approach	Features Used	Machine Learning
CCFinder [17]	✓	×	Suffix-tree and token matching	×	×
Deckard [23]	✓	×	Tree-Matching	×	×
NiCad [18]	✓	×	Textual-Matching	×	×
Yuan and Guo [4]	✓	×	Token-matching	×	×
Komondoor and Horwitz [19]	✓	✓	PDG isomorphism	×	×
Sheneamer and Kalita [21]	✓	×	Hybrid	×	×
SourcererCC [22]	✓	✓	AST isomorphism	×	×
Wang *et al.* [24]	✓	×	Bayesian network	Features based on evolution history, the actual code and the destination of paste	✓
Yang et al [25]	✓	✓	Generalized suffix-tree, TF-IDF vector representation of code, cosine similarity	TF-IDF features	✓
White *et al.* [33]	✓	×	RNNs, RvNNs	Automatic extraction of features from ASTs with tokens represented as vector embeddings	✓
Sæbjörnsen *et al.* [26]	✓	✓	Hashing, l_1-norm among code vectors	Features of normalized assembly Instructions Features	✓
Cesare *et al.* [27]	✓	✓	Naïve Bayes, Tree-based learners	Features based on file names, file sizes, etc.	✓
Wei and Ming [30]	✓	✓	LSTMs	AST and Automatically learned AST-based features and hashcodes	✓
Li *et al.* [28]	✓	✓	Multi-layered perceptron with up to 6 layers	Automatically learned token features	✓
Shalev and Partush [29]	✓	✓	Hashing, vectors of hashes fed to feed-forward ANN	Occurrence counts of smail-sized hashes	✓
Saini *et al.* [32]	✓	×	Siamese twins ANN	AST Features	✓
Tufano *et al.* [34]	✓	✓	RNN, RvNN, autoencoder, Random Forest	I Automatically learned dentifier-, AST-, CFG- and bytecode-based features	✓
Wei and Li [31]	✓	✓	AST-based LSTM and adversarial training	Automatically extracted AST-based features	✓

used are the numbers of variables declared and referenced, maximum nesting depth, number of loops, and number of exceptions thrown.

White el al. [33] used Recursive Neural Networks (RvNNs) to obtain vector embeddings of lexical items, and used Recurrent Neural Networks (RNNs) with modified AST-trees to encode code segments. Encodings of pairs of segments were compared to determine if two blocks are clones. Given a block of code, Tufano *et al.* [34] obtained four different embeddings: identifier-, AST-, bytecode- and CFG-based. To learn embeddings for individual terms, they used an RNN; and for learning encodings for the entire block, they used RvNN-based autoencoders. For CFG-based encoding, they used a graph embedding technique. They computed Euclidean distance among respective embeddings to classify clones. They used Random Forests to combine the individual classifiers' results. A brief summary of some the approaches is reported in Table 1.

We note that the classical algorithms reviewed earlier in this section, mostly do not try to find Type IV clones. However, many of the machine learning based approaches, discussed later in the section, do. The most recent approaches use neural networks of various kinds to classify code clones, with various degrees of success. Some of the approaches use basic feed-forward neural networks. A few approaches use modified LSTMs (Long Short Term Memory), which themselves are a variation of RNNs (Recurrent Neural Networks), with or without the use of RvNNs (Recursive Neural Networks). Others use a variety of other machine learning algorithms, including Naïve Bayes, Decision Trees and Random Forests. Some of the approaches perform end-to-end feature extraction and classification, i.e., they do not need the features to be decided by the designer, and do so automatically. There is usually a trade-off between approaches that use neural networks, such as many layered feed-forward, recurrent or recursive neural networks, compared to non-neural network approaches such as Decision Trees, Random Forests or boosted tree algorithms. Non-ANN algorithms usually need pre-determined features, whereas ANNs can extract features on their own. On the other hand, ANNs are likely to be slower, taking sometimes hours or days for training. Although the recent trend is to use specialized ANN architectures such as RNNs or RvNNs to solve many problems, in this paper, we do not approach with any pre-conceived notions. We compare a large number of algorithms for code clone detection and find that boosted tree algorithms like XGBoost are quite competitive in clone detection. In particular, we extract features from ASTs and PDG and combine the same set of features from both original and copied codes. This fusion of AST and PDG features works extremely well in detecting code clones

of Type-IV in addition to all other types of clones discussed above. The novelty of our work is in comparing the performance of a large number of machine learning techniques, ANN and non-ANN, using such features, and establishing that fusing of AST and PDG features gives competitive results using deep learning as well as boosted tree algorithms.

Next, we discuss in details our machine learning model for effective clone detection.

IV. A MACHINE LEARNING MODEL FOR PAIRWISE CLONE DETECTION

A straightforward approach to determine if two code blocks are semantically similar without necessarily being syntactically similar may proceed as follows: Trim and normalize the two blocks as discussed earlier, obtain the formal semantics of the two blocks using a method alluded to earlier; and, compare the formal semantic representations using Equation 3. However, tools to obtain formal semantics are not readily available. In addition, formal semantic representations are strings themselves, requiring additional string comparisons. It is also unclear that formal semantic representations will add substantially to efficient and effective code clone detection.

Code clone detection has been treated as a pairwise similarity analysis problem, where two blocks are clones if a given block is similar to the given reference block. However, machine learning usually considers individual samples for training and predicts class labels. In any common detection problem, samples used for training and testing have either negative or positive labels. However, in performing clone detection, it is not enough to look at features of one code block to decide whether it is a cloned or not cloned block of code. We need to compare a candidate with another block, usually called a reference block to know if the candidate is a copy of the reference. That is why we consider a <candidate block, reference block> pair as an individual example and assign a label to each such pair. It is always necessary to have a reference based on which one may decide whether a block is cloned. Therefore, it is necessary to consider features of both blocks. We extract relevant characteristics of the blocks by looking at selected portions of them or other associated structures like ASTs and PDGs; these are usually called *features* in the machine learning literature. To apply machine learning to pairwise clone detection, we use features of both the reference and target blocks.

Definition 8 (Pairwise Learning): Given a set of N pairs of training samples, each sample (a pair of blocks) labeled with a clone type depending on their mutual similarity, a classification model can act as a mapping function $f : X \rightarrow Y$, where X is an unknown pair of code blocks and Y is the possible clone type predicted by the model. Training samples are represented as feature vectors, $features(< B_i, B_j >) =< f_1, f_2, \ldots, f_M, C_k >$ of size M, created by combining the features of two different blocks (B_i, B_j) and a clone type, C_k associated with $< B_i, B_j >$, forming a training sample matrix of size $N \times (M + 1)$.

When a block is represented as a set of features, the semantics of a block B_i^n is described as given below:

$$[\![B_i^n]\!] \approx < f_{i1}, \ldots f_{ik} >, \tag{4}$$

where $\approx$ means an approximation. Thus, a block's semantics can be simply represented as a list of features; of course this is not a precise representation of semantic meaning. Equation 2 can now be restated as:

$$\begin{aligned} &semsim(B_i^n, B_j^n) \\ &\quad = semsim(< f_{i1}, \ldots f_{ik} >, < f_{j1}, \ldots f_{jk} >) > \theta. \end{aligned} \tag{5}$$

That is, similarity between two blocks is measured by computing similarity between the two feature based representations.

Thus, instead of using one of the approaches to describe the formal semantics of a program block, we use features of PDGs for semantic representation. We use other features obtained from ASTs as well. In our work, we additionally combine a few so-called traditional features, as discussed later. Next, we discuss our scheme for feature generations.

A. AST AND PDG-BASED NOVEL FEATURES FOR CLONE DETECTION

We pre-process the blocks by trimming and normalizing as discussed earlier. We extract basic characteristics, which we term Traditional Features, like Lines of Code (LOC); and numbers of keywords, variables, assignments, conditional statements and iteration statements [35] used in a given piece of source code. Traditional features alone are inadequate in capturing the syntactic and semantic characteristics of a block.

Syntactic similarity between two blocks of code is also likely to impact upon the similarity in meanings of the blocks, and hence we also parse the blocks into their structural components in terms of Abstract Syntax Tree (AST). Each node of the tree represents a construct occurring in the given source code. Leaf nodes of the tree contain variables used in the code. Unlike majority of published clone detection methods that compare the two syntactic trees directly, we compute certain characteristics or features extracted from the ASTs, which we call *syntactic features*. Figure 2 shows an example AST created by the AST Generator software we use. We traverse the AST in post-order manner and extract only non-leaf nodes containing programming constructs such as Variable Declaration Statements (VDS), While Statements (WS), Cast Expressions, Class Instances, and Method Invocations. Next, we represent frequencies of these programming constructs as AST features in a vector.

The PDG features can be called *semantic* or meaning features. PDGs make explicit both the data and control dependencies for each operation in a program. Data dependencies represent the relevant data flow relationships of a program. Control dependencies represent the essential control flow relationships. A sample PDG derived from a code block is illustrated in Figure 3. Edges represent the order of execution

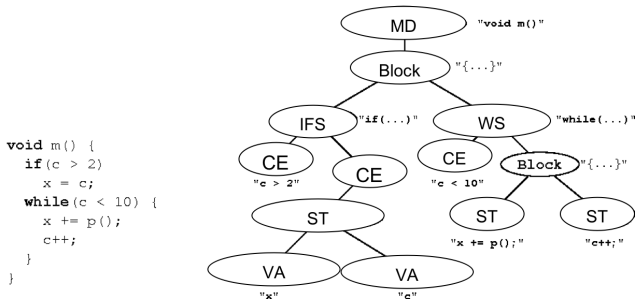


FIGURE 2. Example of AST derived from code block [37]. *MD: MethodDeclaration IFS: IfStatement, WS: WhileStatement, CE: ConditionalExperssion, ST: Statement.*

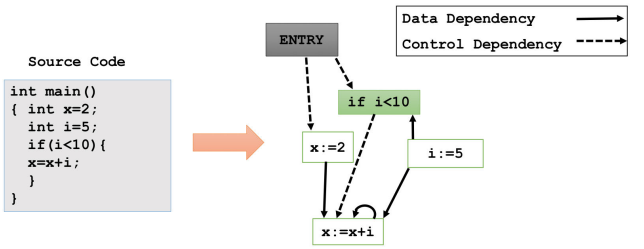


FIGURE 3. Program dependency graph showing control and data dependency among the statements.

of program nodes. The edge of a control flow graph can be used to detect consecutive program nodes as code clones even if they have no data or control dependency. Nodes represent the lines where the corresponding elements are located in the program. Horwitz *et al.* [36] show that PDGs can be used as "adequate" representations of programs and prove that if the PDGs of two graphs are isomorphic, they are strongly equivalent, i.e., they are "programs with the same behavior." We parse the AST, created by an AST generator (`GenerateAST`) further to create an implicit PDG structure and extract features. In other words, we do not construct an explicit PDG but extract the features we could have extracted from an explicit PDG. We use the same post-order traversal of the AST and find the frequencies of various dependency relationships between different constructs. We consider a total of 12 constructs and compute 43 relationships among them up to level three and use them as our PDG or semantic features. For example, the feature *Expr_Assign_Decl*, captures the number of times an Expression occurs, followed by an Assignment, and then followed by a Declaration statements in the given code.

Algorithm 1 describes the feature extraction scheme. $\mathcal{L}_{AST}$ and $\mathcal{L}_{PDG}$ are the lists of pre-specified AST attributes and PDG attributes (please refer to Supplementary material for details). We traverse the non-leaf nodes in the post-order sequence using `PostOrderTokens` and store them in $\mathcal{V}$. We avoid leaf tokens as leaf nodes in AST contain only variables. Frequencies of AST and PDG attributes are stored as features in a vector F. In case the AST attribute `MatchToken` matches each pre-specified AST attribute, we increase the count of that attribute. The method `DependencyFreq` checks for the occurrence of the PDG attribute $\mathcal{L}_{PDG_i}$ in vector $\mathcal{V}$ and returns the frequency of such relationship in $\mathcal{V}$. Please refer to Supplementary materials for the details about the features extracted during the process.

Algorithm 1 AST & PDG Feature Extraction

INPUT : B // Target method block
OUTPUT : $F = \{f_{AST_1}, \ldots, f_{AST_N}, f_{PDG_1}, \ldots, f_{PDG_M}\}$ // Set of N AST and M PDG features
Steps :
$\mathcal{T} \leftarrow \phi$ // AST root node
$\mathcal{L}_{AST} = \{A_1 \ldots A_N\}$ // List of N AST attributes
$\mathcal{L}_{PDG} = \{P_1 \ldots P_M\}$ // List of M PDG attributes
$\mathcal{T} \leftarrow$ `GenerateAST`(B) //Invoking AST generator on B
$\mathcal{V} \leftarrow$ `PostOrderTokens`$(\mathcal{T})$ //Store post order sequence of non-leaf nodes in vector $\mathcal{V}$
//Counting frequency of AST features
for $i = 1 \ldots |\mathcal{L}_{AST}|$ **do**
  for $j = 1 \ldots |\mathcal{V}|$ **do**
    if `MatchToken`$(A_i, \mathcal{V}_j)$ **then**
      $f_{AST_i} = f_{AST_i} + 1$
    end if
  end for
  $F = F \cup f_{AST_i}$
end for
//Counting frequency of PDG features
for $P_i = 1 \ldots |\mathcal{L}_{PDG}|$ **do**
  $f_{PDG_i} \leftarrow$ `DependencyFreq`$(P_i, \mathcal{V})$
  $F = F \cup f_{PDG_i}$
end for
return F

The features of PDG we extract include dependencey information among parts of the code. We extract data dependency features that count the occurrence of declaration, expression, and assignment, in hierarchical ordering, as observed in the PDG. We also extract control dependency features that count the occurrence of the data dependency features. Examples of such features are the number of Assignments that come after Declarations, obtained by counting the occurrence of the assignments which are dependent on declarations; the number of Declarations coming after Control (e.g. $i <$ *count*, *for*, *while*, *if* , *switch*, etc.), obtained by counting the occurrence of the declarations which are dependent on control statements; the number of times a nested iteration occurs; the number of times a nested selection occurs; and so on.

We combine features of ASTs and PDGs for finding syntactic and semantic clones effectively since alone they may not be sufficient. Considering the three types of features we have discussed in this section, we now represent a block in terms of these three types of features. Although it is not strictly semantics any more, we say the "semantics" of a trimmed and normalized code block B_i^n is described as given

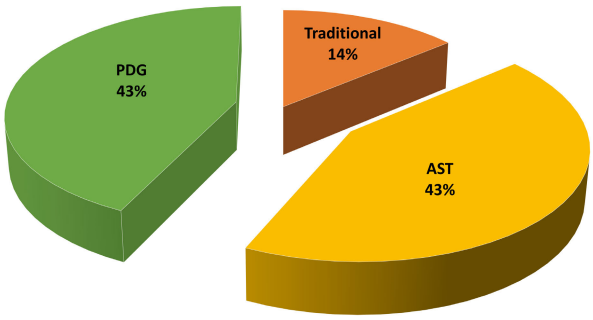


FIGURE 4. Share of different categories of features used.

below:

$$[\![B_i^n]\!] \approx < f_{i1}^t, \ldots f_{ik_t}^t \mid f_{i1}^s, \ldots f_{ik_s}^s \mid f_{i1}^m, \ldots f_{ik_m}^m > . \quad (6)$$

In this equation, we denote the three sets of features with different superscripts: t for traditional features, s for syntactic features, and m for semantic or meaning features, which are actually PDG based features, and separate the three groups with vertical lines, for clear separation. In our work, we generated a total of 100 features, combining the three different types. The distribution of feature categories is shown in Figure 4.

B. FUSION OF BLOCK FEATURES

The fusion of a pair of feature vectors is important to combine the features of the candidate and refernce code blocks and making them a single vector followed by annotation with appropriate class labels. The concept of feature fusion is not new and has been applied successfully in other related problem domains. The fusion of block features has been used in various areas such as text similarity, source code plagiarism, image processing, face detection, and entity resolution and symmetry and object information.

Bilenko and Mooney [38] presented learnable text distance function and vector-space based measure that employs a Support Vector Machine (SVM) for training in duplicate records detection. They used linear combination of features for text similarity. Oyama *et al.* [39] proposed a kernel based method that used combinations of features for matching authors and citations to determine which domain a paper belongs to. Yasaswi *et al.* [40] presented an approach to detect plagiarism in source-code using deep learning features and character-level Recurrent Neural Network (char-RNN). However, their approach only detects copy, partial copy and non-copy. They constructed pairwise features by taking the element-wise difference between individual program feature representations for source code plagiarism. Fu *et al.* [41] proposed a multiple feature fusion in a generalized subspace learning framework. It is to find a general linear subspace in which the cumulative pairwise canonical correlation between every pair of feature sets is maximized after dimension normalization and subspace projection They used multiple feature fusion using subspace learning for face recognition. Atarashi *et al.* [42] used conjunction of features for pairwise classifiers across instances. They then applied the method using support vector machine and simple DNN. They used linear, multiplicative, and distance combinations for entity resolution and symmetry and object information. Lu *et at.* [43] proposed a method that uses feature fusion to represent images for face detection after feature extraction by deep convolutional neural network (DCNN) with SVM classifier. Wang *et al.* [44] proposed a feature fusion algorithm of deep learning and traditional features in image classification which fused the shallow-layer network features, large pre-trained convolutional neural network features and traditional image features together by using a genetic algorithm. Chen *et al.* [45] constructed a novel CNN architecture which contains two independent modules called ''convfusion'' module and asymmetric shortcut connection block, then fine-tuned the hyper parameters in the deep CNN in the second stage and finally applied them to address the problem of Automatic Target Recognition (ATR) using Synthetic Aperture Radar (SAR) images. The combination of a variety of convolution layers and pooling layers plays an influential role in learning robust feature representations. A summary of various techniques that uses feature fusion is shown in Table 2.

To the best of our knowledge, there is no prior use of feature fusion in code clone detection so far. In our work, we combine feature vectors (Equation 6) extracted from a pair of target and reference code blocks to create the training dataset. We fuse the sequences of features from the two different blocks. Although there are three types of features in the description of a block, to simplify the notation, we rewrite Equation 6, without distinguishing among the feature types, as

$$[\![B_i^n]\!] \approx features(B_i) = < f_{i1}, \ldots f_{ik} >, \quad (7)$$

where $k = k_t + k_s + k_m$. Similarly,

$$[\![B_j^n]\!] \approx features(B_j) = < f_{j1}, \ldots f_{jk} > . \quad (8)$$

Given two blocks B_i and B_j, and their clone label C_l, the combined feature vector, $features(< B_i, B_j >)$ can now be represented as a fused feature vector. We fuse the two vectors in three different ways as discussed below.

1) LINEAR COMBINATION

We concatenate the two feature vectors. Simple concatenation gives rise to a fused feature vector of size $2k$. Linear combination looks like follows:

$$features(< B_i, B_j >) = < f_{i1}, \ldots, f_{ik}, f_{j1} \ldots, f_{jk} > . \quad (9)$$

A linear combination results in double the number of features. Linear combination simply lists the features of both candidate clones, giving the machine learning algorithm full freedom to combine the features any way it wants, to compute similarity. Depending on the machine learning algorithm, it may from linear or non-linear combinations of these

TABLE 2. Summary of different methods that uses feature fusion technique.

Method	Domain	Fusion Techniques
Bilenko and Mooney [38]	String similarity	Linear combination.
Oyama *et al.* [39]	Matching authors and citation	Linear, multiplicative, and distance combinations.
Yasaswi *et al.* [40]	Source code plagiarism	Distance combination.
Fu *et al.* [41]	Face recognition	Multiple Feature Fusion.
Atarashi *et al.* [42]	Entity resolution and symmetry	Linear, multiplicative, and distance combinations.
Lu *et al.* [43]	Image processing	A fusion feature method using Deep Convolutional Neural Network (DCNN).
Wang *et al.* [44]	Image processing	A feature fusion using Genetic algorithm.
Chen *et al.* [45]	Image processing	Feature fusion under concatenation pattern and feature fusion under summation pattern.

features. Since linear combination gives rise to a vector of size $2k$, to reduce the size, we experiment with alternative approaches of multiplicative and distance fusion.

2) MULTIPLICATIVE COMBINATION

Here we combine two different feature sequences by multiplying the corresponding feature values:

$$features(< B_i, B_j >) =< f_{i1} * f_{j1}, \ldots, f_{ik} * f_{jk} > . \quad (10)$$

Multiplicative combination simply multiplies the same feature values from the two blocks. If f_{ik} in B_i is 0 (means particular feature not present) and f_{jk} in B_j is 1 (present), multiplication makes the feature value 0 for the combined vector. Hence, in this case, it ignores the importance of f_{ik} and f_{jk} when considering if the block pair belongs to a particular type of clone relationship. Similarly, if the feature values in both the blocks are 0.5, we reduce the combined value of the features further by making it 0.25. When features for two vectors are multiplied component-wise and all these products are added, and the final sum normalized, we get the cosine distance between the two vectors. In this case, our vectors are feature vectors of the two candidate clones. Although we are not performing cosine distance computation directly, we believe a machine learning algorithm may perform the computation of a generalized version of cosine distance in this case or something similar.

3) DISTANCE COMBINATION

Nearness, the opposite of distance, is the most obvious way to calculate the similarity between two block features. We use the absolute difference between two feature values to fuse the features of a pair of blocks.

$$features(< B_i, B_j >) =< |f_{i1} - f_{j1}|, \ldots, |f_{ik} - f_{jk}| > . \quad (11)$$

When pairwise absolute difference of feature values is provided, the machine learning algorithm combines these differences any way as it finds fit to classify the clone pairs. In this case, the machine learning algorithm does not have access to the original feature values, possibly making it less flexible.

C. CLONE DETECTION FRAMEWORK

Our scheme is similar to a traditional machine learning framework. We have two phases, training and testing. In training, we use labelled pairs of cloned blocks from a given hand-curated code clone corpus. All method blocks are detected from the given corpus using lexical and syntactic analysis. We extract method blocks and perform preprocessing, including trimming and normalization. Next, we generate ASTs and PDGs of the blocks and extract features from them [46]. Following Equation 6, we create a complete feature vector for each block by combining traditional, AST and PDG features. We fuse feature vectors of two blocks by using one of the Equations 9, 10 or 11. All the above steps are iterated for all possible pairs of blocks for creating a training dataset for the classification model. For identifying the possible clone type of unlabeled code blocks, we perform the same sequence of steps to create a fused feature vector of the two given blocks and pass it through the classifier for prediction of the possible clone type. Figure 5 demonstrates the work-flow of our approach.

V. EXPERIMENTAL EVALUATION

In this section, we evaluate and compare different machine learning techniques in a unified framework to find which technique works the best to detect all types of clones in large datasets. In our experiments, we use only methods extracted from Java source code as a corpus for training and testing. However, this model is general in nature and can be extended easily to any other high level programming language. Our primary goal is to improve clone detection accuracy for all types of clones with a special emphasis on semantic clone detection. We use a number of existing classification algorithms and compare the effectiveness of the proposed framework with state-of-the-art detection methods based on their reported results.

A. DATASETS

We use IJaDataset 2.0 [47], a large inter-project Java repository containing source code from 25,000 open-source projects, with 3 million source files, 250 million lines of code, from SourceForge and Google Code. This benchmark was built by mining IJaDataset for functions. The published version of the benchmark considers 44 functions [48].

For this experiment, we consider all types of clones in IJaDataset 2.0 that are 6 lines or 50 tokens or longer, which is the standard minimum clone size for benchmarking [11], [22]. There is no agreement on when a clone is no longer syntactically similar, and many authors claim that it is also hard to separate the Type-III and Type-IV clones in the IJaDataset [47]. As a result, some prior researchers have divided Type-III and Type-IV clones into four classes based on their syntactic similarity [22] as follows: Very Strongly Type-III

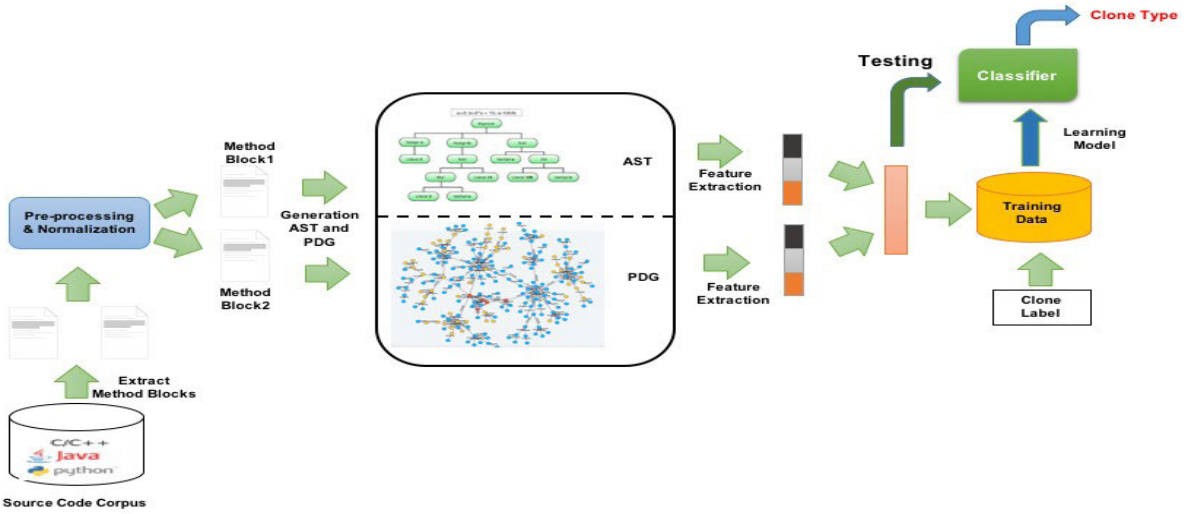


FIGURE 5. Workflow of the proposed clone detection framework.

TABLE 3. Brief description of our java code clone corpora.

Dataset	Paired Blocks	Type-I and II	Type-III	Type-IV	False Positive	Agreement(%)
IJaDataset2.0	106,736	≈ 19, 533(18.73%)	≈ 46.750(43.8%)	≈ 19, 533(18.73%)	≈ 19, 533(18.73%)	-
EIRC	870	32	394	146	298	76
Sample_j2sdk1.4.0-javax-swing	800	200	282	118	200	76
Sample_eclipse-jdtcore	800	200	200	169	231	69
eclispe-ant	787	118	392	66	211	60
netbeans-javadoc	452	54	146	39	213	62
Suple	152	30	59	25	38	75

(VST3) clones are ones that have a syntactic similarity in the range [90% 100%), Strongly Type-III (ST3) in [70% - 90%), Moderately Type-III in [50% - 70%) and Weakly Type-III/Type-IV (WT3/4) in (0%-50%], where (means exclusive and] means inclusive range.

A majority of existing clone datasets used in prior papers are incomplete in nature. They usually avoid labeling semantic code clones. Some of the publicly available datasets are *eclipse-ant*, *eclipse-jdtcore*, *netbeans-javadoc* and *j2sdk14.0-javax-swing*. Moreover, the original datasets contain small numbers of instances of specific types, making them difficult to use for machine learning. To overcome this situation, we extract additional method blocks from the original source codes and label them using a semi-supervised labelling method [49]. The details of the datasets are given in Table 3. In the table, the second column indicates how many paired-blocks we extracted to expand the existing datasets. "Agreement" refers to the probability of reliability between observers or raters. We compute Kappa statistic [50] agreement between every two observers' decisions using Equation 12 and take the average probability of agreement between all the raters and report the same in the Table 3:

$$\kappa = \frac{p_o - p_e}{1 - p_e}, \tag{12}$$

where, p_o is the relative observed agreement among raters and p_e is the hypothetical probability of chance agreement.

We also report, in the table, the number of labeled samples of a particular type of clone (I, II, III or IV) present in the dataset.

B. CLASSIFICATION MODELS

We train and test our proposed framework using fifteen classification models, starting from the popularly used Naïve Bayes [51] model to a recently published gradient boosting tree model, XGBoost (eXtreme Gradient Boosting) [52]. While selecting the various classification models, we try to keep a balance among different learning models including probabilistic and non-probabilistic, generative and discriminative, linear and non-linear, regression, decision trees and distance based models. We also try both traditional and modern approaches, as well as individual and ensemble approaches.

Naïve Bayes [51] is a simple probabilistic classifier based on Bayes' rule. Linear Discriminant Analysis (LDA) [53] is commonly used as a dimensionality reduction technique in pre-processing for pattern-classification and machine learning applications and can be used as a classifier also. Support Vector Machine (SVM) [54] is a maximum margin classification model. LogitBoost [55] is a boosting classification algorithm. LogitBoost and AdaBoost are close to each other in that both perform additive logistic regression. Instance Based Learner (IBK) [56] is similar to a k-Nearest Neighbor algorithm. In addition, we use several tree ensemble models including Extra Trees [57], Rotation Forest [58] coupled with

TABLE 4. Different classification techniques used.

Classifiers	Model Characteristics	Platform
Naïve Bayes	Probabilistic, makes independence assumption among features	Weka 3.8
LDA	Finds a linear combination of features as separator between classes	Weka 3.8
LIBLINEAR/SVM	Linear maximum margin classifier	Weka 3.8
Sequential Minimal Optimization (SMO)	Quadratic programming solver for SVMs	Weka 3.8
IBK	K nearest-neighbor classifier, can choose k using cross-validation	Weka 3.8
J48	An implementation of C4.5 decision tree classifier	Weka 3.8
Random Tree	A decision tree classifier that uses k random attributes at each node	Weka 3.8
Extra Tree	A decision tree classifier that, works with numeric attributes also	Weka 3.8
Boostrap aggregation (Bagging)	Ensemble classifier that creates a classifier from separates samples of the training dataset	Weka 3.8
LogitBoost	A statistical implementation of Adaboost, a meta-learning algorithm	Weka 3.8
Random Supspace	Ensemble classifier that creates multiple decision trees constructed by randomly chosen features	Weka 3.8
Random Committee	An ensemble of randomize base classifiers	Weka 3.8
Rotation Forest	An ensemble of classifiers created by making k subsets of features, running PCA on each subset, and keeping all principal components	Weka 3.8
Random Forest	Ensemble of decision trees	Weka 3.8
XGBoost	Ensemble classifier that creates gradient boosted decision trees, each with an associated objective function, and a regularizer that is optimized,	R 3.3.0
Convolutional Neural Network (CNN)	Deep Learning	Weka 3.8

Principal Components Analysis (PCA), Random Forest [59] and Random Committee [60]. Bagging [61] is an ensemble meta-estimator that fits base classifiers each on random subsets of the original dataset and then aggregates their individual predictions. We also use decision tree algorithms such as J48 [62] and Random tree [63] for our experimentations. Random Subspace [64] selects random subsets of the available features to be used in training the individual classifiers in an ensemble. XGBoost [52] is a fast and accurate boosting tree model proposed recently. At this time, any classification task is incomplete without the use of deep learning based classification. Deep learning models with varying configurations have been successfully applied in different application domains [65] [66]. A very commonly used model is **Convolutional Neural Network (CNN)**. With CNNs, it is necessary to try many different configurations to see what works best for the problem at hand. In Computer Vision, where CNNs shine [66], the models are fed a 2-D array of pixels and the neural networks perform end-to-end processing. The CNN extracts progressively higher levels of features on its own through the use of several convolution layers. The problem of clone detection, the focus of this paper, is more akin to problems faced in natural language processing (NLP) because both deal with string-based entities. In NLP, CNNs and other deep learning methods work well when the words are first converted to embedding using a method such as word2vec [67]. In this paper, we do not obtain embeddings for programming language constructs and keywords; we work with the 100 features we described earlier.

We use *WekaDeeplearning4j*[1] to implement CNN. It allows arbitrary-depth multi-layer network with certain degree of flexibility in selecting types of weight initialization, loss function, gradient descent algorithm, etc. We experimented with several different architectures and we briefly describe the one that works best. It is a simple model with three convolutional layers stacked on top of each other. Each convolutional layer is followed by a max pooling layer. The last pooling layer is followed by a fully connected layer with ReLU activation and then a softmax layer. We arrange our input data as a $100 \times 1 \times 1$ tensor and feed into convolution layer $\rightarrow$ max pooling layer, and so on.

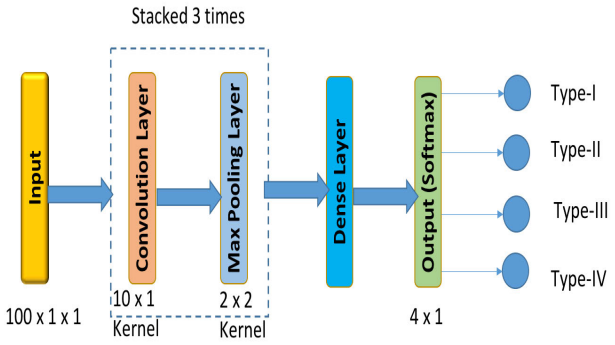


FIGURE 6. Convolutional neural networks architecture for code clone classification.

Our input is a vector of 100 features. We reshape our input from 1-D to 3-D using the *reshape* function in wekaDeeplearning4j package itself. The convolution is applied on the input data using a convolution filter to produce a feature map. We use 10×1 filters. The convolution operation is performed by sliding this filter over the input. We move the convolution filter by 1 stride at each step. After the convolution operation, pooling is performed to reduce the dimensionality. Pooling is performed with 2×2 windows with stride 2. In the fully connected layer, the data is flattened (one dimension) and the next layer is the output layer, which consists of 4 nodes corresponding to 4 classes. The architecture of our CNN implementation is shown in Figure 6.

The classification models used in our work are summarized in Table 4.

We represent a pair instance as a vector as explained in section IV-B. Clones of different types are detected using one of the classification algorithms. We compare the outcomes of

[1] https://github.com/Waikato/wekaDeeplearning4j

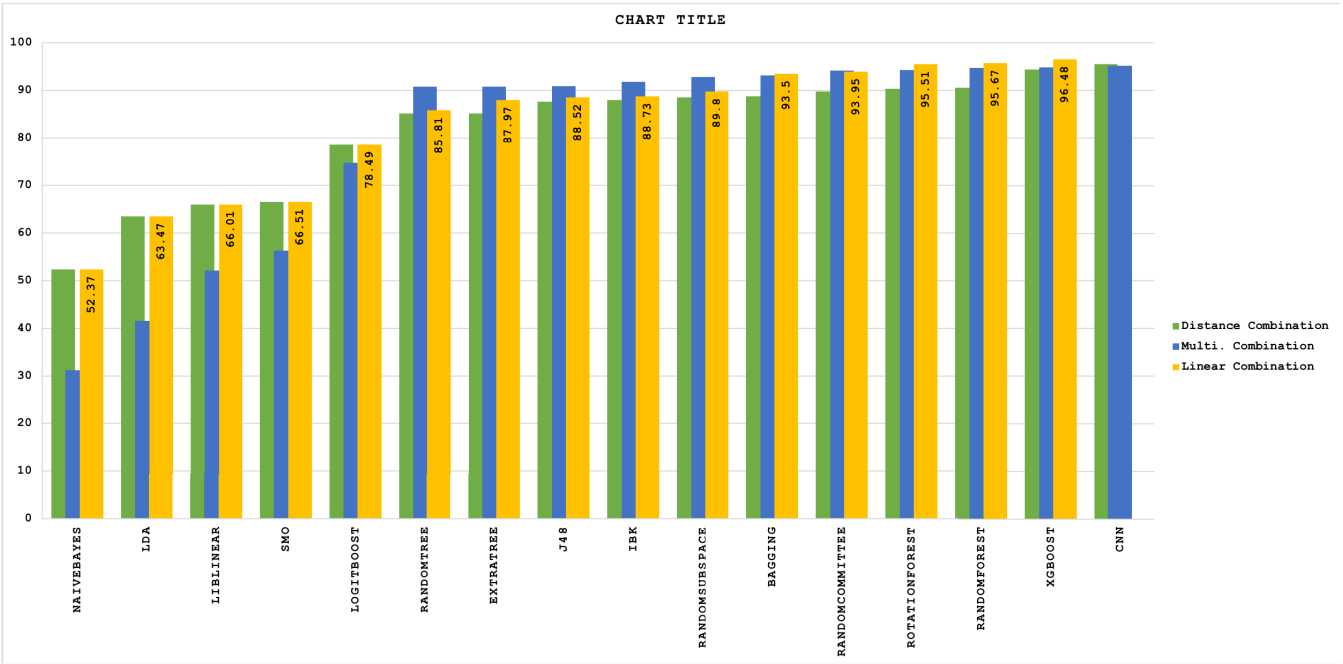


FIGURE 7. Performance of all the candidate classifiers with different feature fusions on IJaDataset.

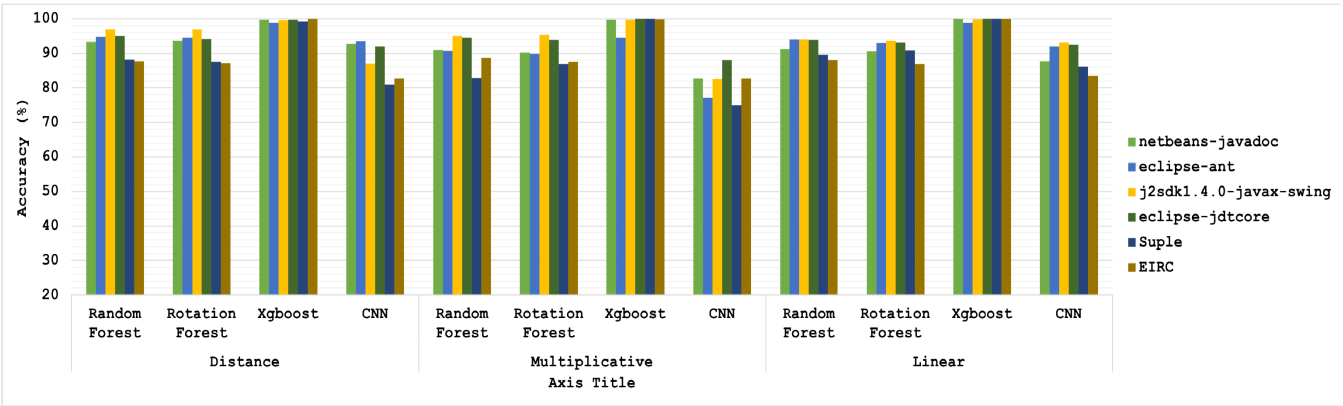


FIGURE 8. Performance of random forest, rotation forest, XGBoost, and CNN in six different datasets.

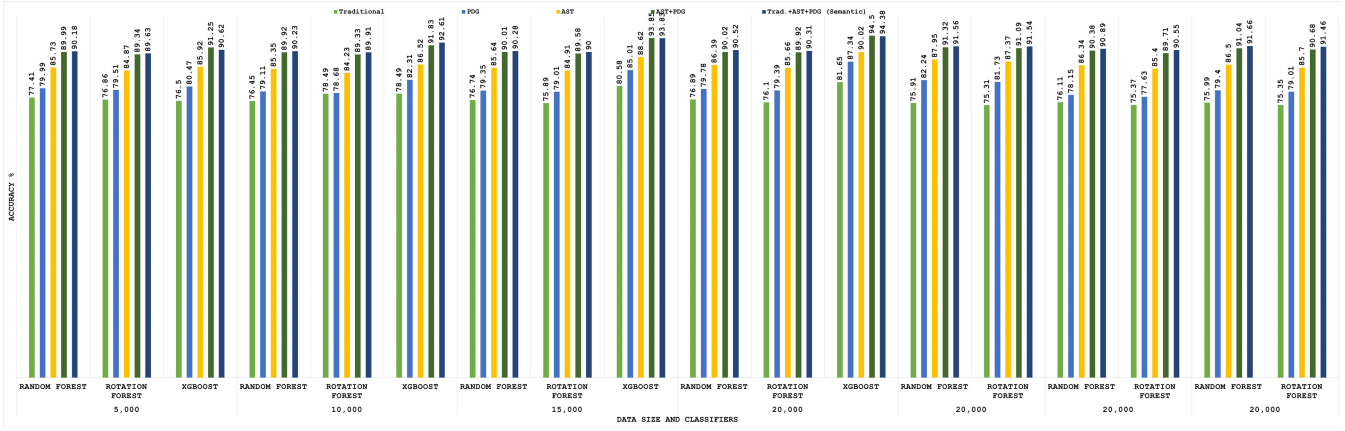
all the classifiers discussed above, as our main emphasis has been on effective feature generation. Classifiers are trained and tested using cross-validation with 10 folds. We ensure balance between match and non-match classes in each fold and the same as in the overall dataset.

C. EVALUATION

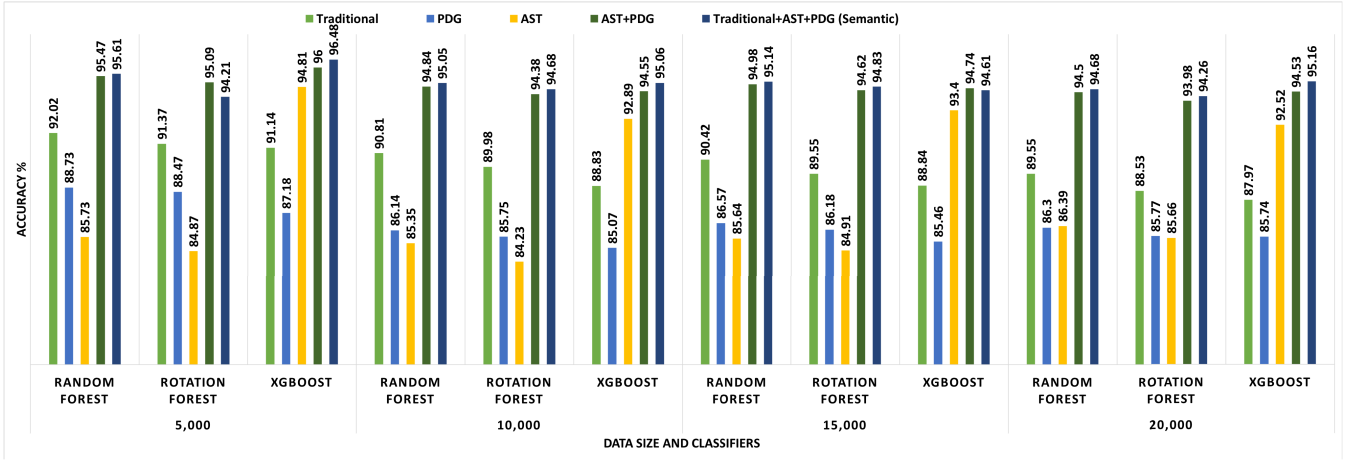
We generate extensive results to assess the robustness of our proposed model in detecting semantic clones along with all other type of clones. We experiment with a varying number of features and with different data to show that our features are able to achieve high detection accuracy. Due to space limitations, we report only best performing classifiers for most of the experiments and compare them with state-of-the-art clone detection methods. However, for more results, one can refer to the Supplementary materials. To generate AST from a given block in order to extract features, we use Eclipse Java Development Tools (JDT).

1) PERFORMANCE OF DIFFERENT CLASSIFIERS

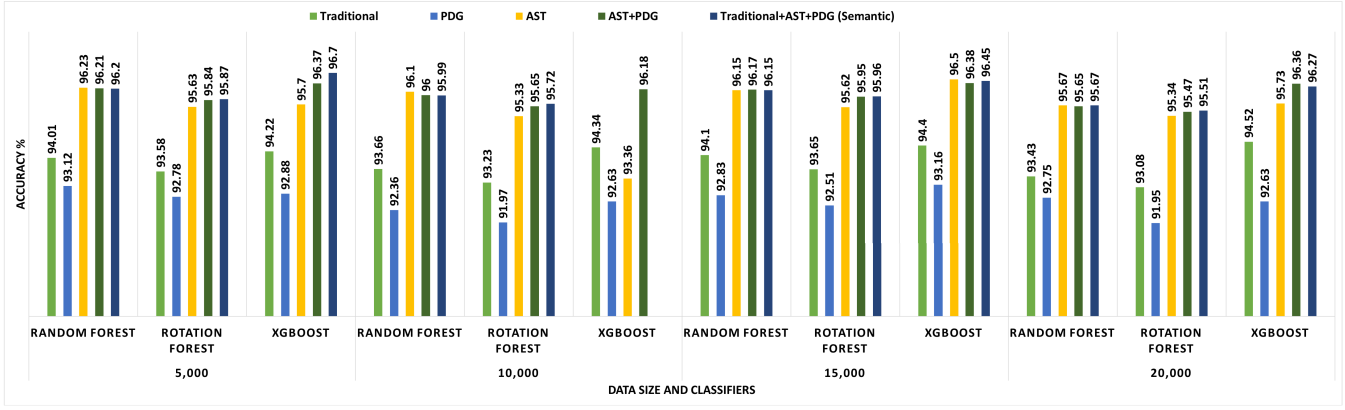
We randomly select 20K pair instances from the IJaDataset. To compare the three feature fusion methods and the performance of classifiers, we run all the classifiers three times. All the candidate models are trained and tested using 10 fold cross-validation, where we ensure that the ratio between match and non-match classes is the same in each fold and the same as in the overall dataset. Figure 7 shows the comparison of all fifteen classifiers using linear, multiplicative, and distance combinations respectively on IJaDataset. Experimental results show that the tree ensemble methods such



(a) *Distance Combination*



(b) *Multiplicative Combination*



(c) *Linear Combination*

FIGURE 9. **Performance of three best classifiers with syntactic and semantic features on IJaDataset dataset.**

as Rotation Forest, Random Forest and XGBoost achieve better outcomes among all the classifiers. This is because tree ensemble approaches create many trees with samples and random attributes. XGBoost has high performance as it has a regularization component in the loss function to reduce overfitting. Due to heavy computational time requirements by CNN, we do not apply CNN on the IJaDataset.

However, we apply CNN on six different clone corpora which are relatively small in size and compare the results with the tree based ensemble methods (see Figure 8), which appears to be best performer in the large of dataset, IJaDataset. Our observation is that Random Forests and XGBoost are highly competitive with CNNs. Remarkably, the performance of CNN is relatively poor in this

context, though we report best results produced by CNN after considerable parameter tuning. The possible reason of low performance may be due to relatively low dimension of the dataset used, though we use 100 features for our experiments. Once again, the deep learning model usually extracts features automatically. In our case, we have performed feature extraction on our own. This may be a cause for lower performance as well. However, the performance of XGBoost is vastly superior in all six small datasets.

2) VARYING DATA SIZE AND FEATURE TYPES

We assess the importance of combining traditional, AST and PDG features in different combinations and report the results on IJaDataset in Figure 9. We create four subsets of IJaDataset using 5K, 10K, 15K and 20K instances from each class. Results produced by three best performing classifiers reported above with varying data sizes, show that the performance of the classifiers improves substantially as we combine both syntactic and semantic features to detect clones. Interestingly, the performance of the classifiers using semantic features is consistent irrespective of data sizes and fusion methods. We also observe that a linear combination produces better results than distance and multiplicative combinations for all sizes of data. Linear combination works better in comparison to other two methods, because linear combination keeps the original feature values unchanged, which the other two methods do not. Multiplicative and distance combination may hamper the classifiers because they may find such combinations unsuitable to work with effectively.

3) EXPERIMENTING WITH VARYING FEATURE SIZES

We perform two different kinds of experiments with varying numbers of features, selecting equal numbers of features from each feature type (Traditional, AST and PDG) and using feature selection methods (Figure 11). The intention behind such experiments is to show the significance of our proposed features in achieving better accuracy, and that it is not by chance. The growing learning curve (Figure 10) clearly indicates that the detection accuracy improves with the increase in the numbers of features. We also notice that XGBoost using multiplicative combination achieves higher performance than others. Feature selection or extraction [68], [69] is a predominant preprocessing step in many traditional machine learning based pattern recognition tasks, although deep learning models are end-to-end, performing automatic feature selection. We use two feature selection methods namely Gain Ratio [70] and Information Gain [71]. For each experiment, we use different sizes of the feature sets ranked by the feature selection algorithms. Similar to the learning curve based on randomly selected feature sets, judiciously selected feature sets also show a growing trend in performance. This further establishes the fact that our features are crucial in deriving high accuracy detection results.

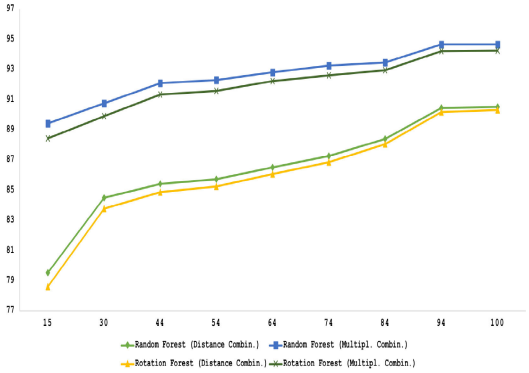
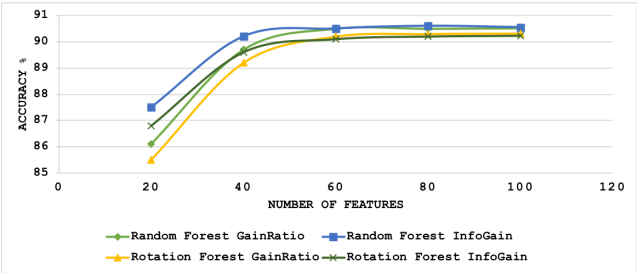
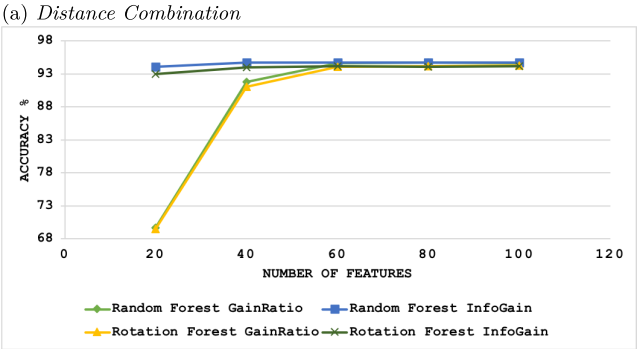


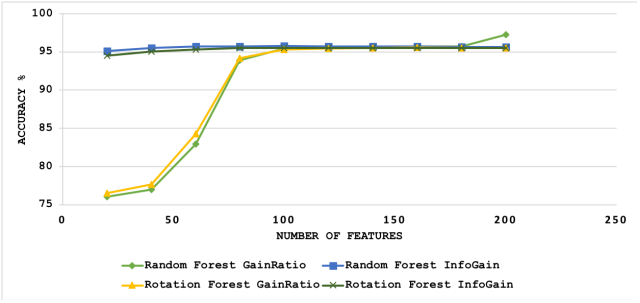
FIGURE 10. Learning Curve: performance of random and rotation forest with varying features on IJaDataset dataset.



(a) *Distance Combination*



(b) *Multiplicative Combination*

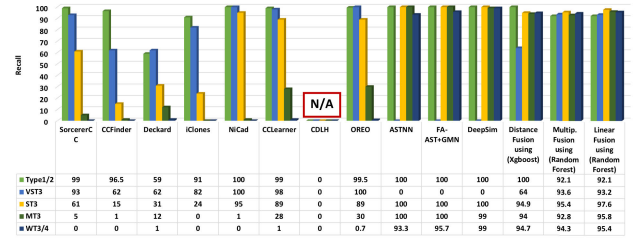


(c) *Linear Combination*

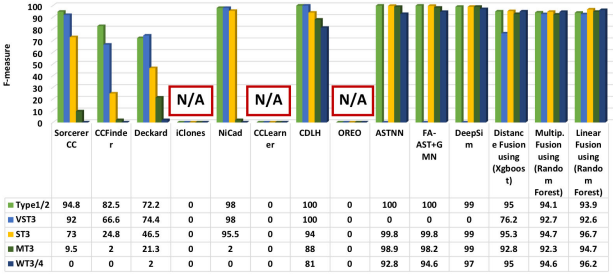
FIGURE 11. Performance of random forest and rotation forest with varying features using gain ratio and InfoGain feature selection algorithms on IJaDataset dataset.

4) PERFORMANCE COMPARISON

We compare the performance of our method with contemporary clone detection methods, using their reported results on IJaDataset. Different papers have reported a range of Precision, Recall and F-score values for different clone detection methods. We show the maximum value of the reported range



(a) *Recall Assessments*



(b) *F-measure Assessments*

FIGURE 12. Performance comparison of different detection methods with respect to different assessment metrics on IJaDataset dataset.

when we report their results. Interestingly, a majority of the detection methods are incapable of detecting semantic clones or Type-IV clones. Figure 12 shows comparison of our results with the state-of-the-art detectors based on recall and F-score. From the results, it is evident that NiCad performs better with respect to all other methods in detecting Type-I/II, VST3, and ST3 clones based on the F-measure metric. We report only results of XGBoost and Random Forest with various fusion types. Results clearly show that our method is effective in detecting Type-IV clones along with other clone types in comparison to the other methods.

VI. THREATS TO VALIDITY

Since we consider only granularity of method level clones, we may miss overlapping clones in Java classes. However, most clones in Java code fragments are represented at a method level. To generate AST and PDG from a given block in order to extract features, we use Eclipse Java Development Tools (JDT). The datasets are restricted to Java-based clones. We plan to involve other programming languages in the future.

VII. CONCLUSION

Semantic code clone detection is a challenging task and needs automatization, especially, because the amount and sizes of complex software written are increasing. We propose a machine learning framework for automatic detection of large numbers of code clones. We use, for the first time, a combination of traditional, AST and PDG features in machine learning instead of using them for computing graph isomorphism. We captured the syntax of program codes using AST program features, and the semantics of program codes using PDG features. We use fifteen classification models to obtain their relative performance using our features. We performed an extensive set of experiments to show that our machine learning framework is able to identify the technique that can detect clones the best. Experimental results clearly indicate that our proposed features are highly valuable in achieving high detection accuracy.

As a part of our future endeavor, we would like to extend our work to achieve further improvements, for example, by using features of Java byte and assembly codes obtained by compiling Java programs. We also intend to create token embeddings from the datasets to use them in deep learning. We also plan to explore various deep learning architectures such as CNNs and RNNs.

ACKNOWLEDGMENT

The work reported in this paper was supported by Jazan University, Saudi Arabia. The authors would like also to thank members of the LINC Lab at University of Colorado at Colorado Springs for helpful discussions and suggestions.

REFERENCES

[1] B. S. Baker, "Finding clones with dup: Analysis of an experiment," *IEEE Trans. Softw. Eng.*, vol. 33, no. 9, pp. 608–621, Sep. 2007.
[2] C. K. Roy and J. R. Cordy, "An empirical study of function clones in open source software," in *Proc. 15th Work. Conf. Reverse Eng.*, Oct. 2008, pp. 81–90.
[3] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queens School Comput.*, vol. 541, no. 115, pp. 64–68, 2007.
[4] Y. Yuan and Y. Guo, "CMCD: Count matrix based code clone detection," in *Proc. 18th Asia–Pacific Softw. Eng. Conf.*, Dec. 2011, pp. 250–257.
[5] S. Schulze and D. Meyer, "On the robustness of clone detection to code obfuscation," in *Proc. 7th Int. Workshop Softw. Clones (IWSC)*, May 2013, pp. 62–68.
[6] M. R. Farhadi, B. C. M. Fung, P. Charland, and M. Debbabi, "BinClone: Detecting code clones in malware," in *Proc. 8th Int. Conf. Softw. Secur. Rel. (SERE)*, Jun. 2014, pp. 78–87.
[7] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *Proc. 6th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng. (ESEC-FSE)*, 2007, pp. 55–64.
[8] F. A. Fontana, M. Zanoni, A. Ranchetti, and D. Ranchetti, "Software clone detection and refactoring," *ISRN Softw. Eng.*, vol. 2013, pp. 1–8, Mar. 2013.
[9] W. H. Gomaa and A. A. Fahmy, "A survey of text similarity approaches," *Int. J. Comput. Appl.*, vol. 68, no. 13, pp. 13–18, Apr. 2013.
[10] N. Gali, R. Mariescu-Istodor, D. Hostettler, and P. Fränti, "Framework for syntactic string similarity measures," *Expert Syst. Appl.*, vol. 129, pp. 169–185, Sep. 2019.
[11] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Trans. Softw. Eng.*, vol. 33, no. 9, pp. 577–591, Sep. 2007.
[12] W. J. Ewens and G. R. Grant, *Statistical Methods in Bioinformatics: An Introduction*. Springer, 2006.
[13] P. Pevzner, *Computational Molecular Biology: An Algorithmic Approach*. Cambridge, MA, USA: MIT Press, 2000.
[14] A. Kulkarni and R. Metta, "A code obfuscation framework using code clones," in *Proc. 22nd Int. Conf. Program Comprehension (ICPC)*, 2014, pp. 295–299.
[15] C. A. Gunter, *Semantics of Programming Languages: Structures and Techniques*. Cambridge, MA, USA: MIT Press, 1992.
[16] G. Winskel, *The Formal Semantics of Programming Languages: An Introduction*. Cambridge, MA, USA: MIT Press, 1993.
[17] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002.
[18] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proc. 16th IEEE Int. Conf. Program Comprehension*, Jun. 2008, pp. 172–181.

[19] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proc. Int. Static Anal. Symp.* Berlin, Germany: Springer, 2001, pp. 40–56.

[20] Y. Higo and S. Kusumoto, "Code clone detection on specialized PDGs with heuristics," in *Proc. 15th Eur. Conf. Softw. Maintenance Reeng. (CSMR)*, Mar. 2011, pp. 75–84.

[21] A. Sheneamer and J. Kalita, "Code clone detection using coarse and fine-grained hybrid approaches," in *Proc. IEEE 7th Int. Conf. Intell. Comput. Inf. Syst. (ICICIS)*, Dec. 2015, pp. 472–480.

[22] V. Saini, H. Sajnani, J. Kim, and C. Lopes, "SourcererCC and SourcererCC-I: Tools to detect clones in batch mode and during software development," in *Proc. 38th Int. Conf. Softw. Eng. Companion*, May 2016, pp. 597–600.

[23] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proc. 29th Int. Conf. Softw. Eng. (ICSE)*, May 2007, pp. 96–105.

[24] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei, "Can I clone this piece of code here?" in *Proc. 27th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2012, pp. 170–179.

[25] J. Yang, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Classification model for code clones based on machine learning," *Empirical Softw. Eng.*, vol. 20, no. 4, pp. 1095–1125, Aug. 2015.

[26] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proc. 18th Int. Symp. Softw. Test. Anal. (ISSTA)*, 2009, pp. 117–128.

[27] S. Cesare, Y. Xiang, and J. Zhang, "Clonewise–detecting package-level clones using machine learning," in *Proc. Int. Conf. Secur. Privacy Commun. Syst.* Cham, Switzerland: Springer, 2013, pp. 197–215.

[28] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "CCLearner: A deep learning-based clone detection approach," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2017, pp. 249–260.

[29] N. Shalev and N. Partush, "Binary similarity detection using machine learning," in *Proc. 13th Workshop Program. Lang. Anal. Secur.*, 2018, pp. 42–47.

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

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

[32] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. Lopes, "Oreo: Detection of clones in the twilight zone," 2018, *arXiv:1806.05837*. [Online]. Available: http://arxiv.org/abs/1806.05837

[33] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, Aug. 2016, pp. 87–98.

[34] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *Proc. 15th Int. Conf. Mining Softw. Repositories*, May 2018, pp. 542–553.

[35] E. Kodhai, S. Kanmani, A. Kamatchi, R. Radhika, and B. V. Saranya, "Detection of type-1 and type-2 code clones using textual analysis and metrics," in *Proc. Int. Conf. Recent Trends Inf., Telecommun. Comput.*, Mar. 2010, pp. 241–243.

[36] S. Horwitz, J. Prins, and T. Reps, "On the adequacy of program dependence graphs for representing programs," in *Proc. 15th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, 1988, pp. 146–157.

[37] E. Söderberg, T. Ekman, G. Hedin, and E. Magnusson, "Extensible intraprocedural flow analysis at the abstract syntax tree level," *Sci. Comput. Program.*, vol. 78, no. 10, pp. 1809–1827, 2013.

[38] M. Bilenko and R. J. Mooney, "Adaptive duplicate detection using learnable string similarity measures," in *Proc. 9th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2003, pp. 39–48.

[39] S. Oyama and C. D. Manning, "Using feature conjunctions across examples for learning pairwise classifiers," in *Proc. Eur. Conf. Mach. Learn.* Berlin, Germany: Springer, 2004, pp. 322–333.

[40] J. Yasaswi, S. Purini, and C. Jawahar, "Plagiarism detection in programming assignments using deep features," in *Proc. 38th SIGCSE Tech. Symp. Comput. Sci. Educ.*, 2007, p. 34.

[41] Y. Fu, L. Cao, G. Guo, and T. S. Huang, "Multiple feature fusion by subspace learning," in *Proc. Int. Conf. Content-Based Image Video Retr. (CIVR)*, 2008, pp. 127–134.

[42] K. Atarashi, S. Oyama, M. Kurihara, and K. Furudo, "A deep neural network for pairwise classification: Enabling feature conjunctions and ensuring symmetry," in *Proc. Pacific-Asia Conf. Knowl. Discovery Data Mining*. Cham, Switzerland: Springer, 2017, pp. 83–95.

[43] X. Lu, X. Duan, X. Mao, Y. Li, and X. Zhang, "Feature extraction and fusion using deep convolutional neural networks for face detection," *Math. Problems Eng.*, vol. 2017, pp. 1–9, Jan. 2017.

[44] Y. Wang, B. Song, P. Zhang, N. Xin, and G. Cao, "A fast feature fusion algorithm in image classification for cyber physical systems," *IEEE Access*, vol. 5, pp. 9089–9098, 2017.

[45] S.-Q. Chen, R.-H. Zhan, J.-M. Hu, and J. Zhang, "Feature fusion based on convolutional neural network for SAR ATR," in *Proc. ITM Web Conf.*, vol. 12, 2017, p. 05001.

[46] A. Sheneamer and J. Kalita, "Semantic clone detection using machine learning," in *Proc. 15th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2016, pp. 1024–1028.

[47] 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 *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, Sep. 2014, pp. 476–480.

[48] I. Keivanloo, F. Zhang, and Y. Zou, "Threshold-free code clone detection for a large-scale heterogeneous java repository," in *Proc. IEEE 22nd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, Mar. 2015, pp. 201–210.

[49] A. Sheneamer, H. Hazazi, S. Roy, and J. Kalita, "Schemes for labeling semantic code clones using machine learning," in *Proc. 16th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2017, pp. 981–985.

[50] A. J. Viera and J. M. Garrett, "Understanding interobserver agreement: The kappa statistic," *Family Med.*, vol. 37, no. 5, pp. 360–363, 2005.

[51] G. H. John and P. Langley, "Estimating continuous distributions in Bayesian classifiers," in *Proc. 11th Conf. Uncertainty Artif. Intell.* San Mateo, CA, USA: Morgan Kaufmann, 1995, pp. 338–345.

[52] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," 2016, *arXiv:1603.02754*. [Online]. Available: http://arxiv.org/abs/1603.02754

[53] D. F. Morrison, "Multivariate analysis of variance," in *Encyclopedia of Biostatistics*, vol. 5. 2005.

[54] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "LIBLINEAR: A library for large linear classification," *J. Mach. Learn. Res.*, vol. 9, pp. 1871–1874, Jun. 2008.

[55] J. Friedman, T. Hastie, and R. Tibshirani, "Additive logistic regression: A statistical view of boosting (with discussion and a rejoinder by the authors)," *Ann. Statist.*, vol. 28, no. 2, pp. 337–407, Apr. 2000.

[56] D. W. Aha, D. Kibler, and M. K. Albert, "Instance-based learning algorithms," *Mach. Learn.*, vol. 6, no. 1, pp. 37–66, Jan. 1991.

[57] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Mach. Learn.*, vol. 63, no. 1, pp. 3–42, Apr. 2006.

[58] J. J. Rodriguez, L. I. Kuncheva, and C. J. Alonso, "Rotation forest: A new classifier ensemble method," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 28, no. 10, pp. 1619–1630, Oct. 2006.

[59] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[60] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*. San Mateo, CA, USA: Morgan Kaufmann, 2005.

[61] L. Breiman, "Bagging predictors," *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, Aug. 1996.

[62] S. L. Salzberg, "C4.5: Programs for machine learning by J. Ross Quinlan. Morgan Kaufmann publishers, Inc., 1993," *Mach. Learn.*, vol. 16, no. 3, pp. 235–240, Sep. 1994.

[63] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Tech. Rep., 1998.

[64] T. K. Ho, "The random subspace method for constructing decision forests," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 20, no. 8, pp. 832–844, Aug. 1998.

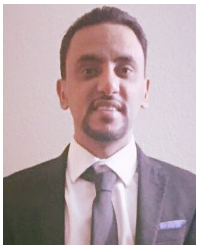
[65] X. Yang, W. Liu, D. Tao, and J. Cheng, "Canonical correlation analysis networks for two-view image recognition," *Inf. Sci.*, vols. 385–386, pp. 338–352, Apr. 2017.

[66] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[67] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. Adv. Neural Inf. Process. Syst.*, 2013, pp. 3111–3119.

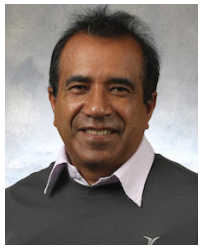
[68] W. Liu, X. Yang, D. Tao, J. Cheng, and Y. Tang, "Multiview dimension reduction via hessian multiset canonical correlations," *Inf. Fusion*, vol. 41, pp. 119–128, May 2018.

[69] Y.-H. Yuan, Y. Li, X.-B. Shen, Q.-S. Sun, and J.-L. Yang, "Laplacian multiset canonical correlations for multiview feature extraction and image recognition," *Multimedia Tools Appl.*, vol. 76, no. 1, pp. 731–755, Jan. 2017.
[70] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Amsterdam, The Netherlands: Elsevier, 2014.
[71] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, 1986.



ABDULLAH SHENEAMER received the B.Sc. degree in computer science from King Abdulaziz University, Saudi Arabia, in 2008, and the M.Sc. and Ph.D. degrees in computer science from the University of Colorado at Colorado Springs, USA, in 2012 and 2017, respectively. He is currently an Assistant Professor of computer science with Jazan University, Saudi Arabia. His current work focuses on software clone and malware and code obfuscation detection using machine learning. He has published several articles in reputed international journals and several papers in international conferences. His research interests include data mining, machine learning, malware analysis, and software engineering. He is a Reviewer of various journals, including IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, *Information Sciences*, and IEEE ACCESS, and a Senior Meta-Reviewer of IEEE International Conference on Machine Learning and Applications.

SWARUP ROY received the M.Tech. and Ph.D. degrees in computer science and engineering from Tezpur University, India. He worked as a Postdoctoral Fellow with the University of Colorado at Colorado Springs and the Indian Institute of Technology Guwahati. He is currently an Associate Professor of computer science with Sikkim University (Central), India. He has published a number of articles in peer-reviewed international journals and conference proceedings and authored two books. His research interests include data mining, machine learning and their applications to computational biology, social network analysis, smartphone security, and code clone detection. He is a member of the technical committees of various reputed international conferences and the Co-Chair of the Biological Modelling Track ACM-BCB 2017.



JUGAL KALITA received the B.Tech. degree from the Indian Institute of Technology Kharagpur, Kharagpur, India, the M.S. degree from the University of Saskatchewan, Canada, and the M.S. and Ph.D. degrees from the University of Pennsylvania. He is currently a Professor of computer science with the University of Colorado at Colorado Springs. He is the author of more than 200 research papers in reputed conferences and articles in journals, and has authored four books. His research interest includes machine learning and its applications to areas, such as natural language processing, cybersecurity, and bioinformatics. He has received multiple NSF grants and regulary serves on NSF Panels.