Schemes for Labeling Semantic Code Clones using Machine Learning

Abdullah Sheneamer††, Hanan Hazazi‡, Swarup Roy‡§, and Jugal Kalita†

†asheneam@uccs.edu, ‡hhazazi@regis.edu, †swarup@nehu.ac.in, §jkalita@uccs.edu

*Faculty of Computer Science & Information Systems, Jazan University, Jazan 45142 KSA
†College of Engineering & Applied Science, University of Colorado, Colorado Springs, CO 80918 USA
‡College of Computer & Information Sciences, Regis University, Denver, CO 80221 USA
§Department of Information Technology, North-Eastern Hill University, Shillong, 793022 INDIA

Abstract—Machine learning approaches built to identify code clones fail to perform well due to insufficient training samples and have been restricted only up to Type-III clones. A majority of the publicly available code clone corpora are incomplete in nature and lack labeled samples for semantic or Type-IV clones. We present here two schemes for labeling all types of clones including Type-IV clones. We restrict our study to Java code only. First, we use an unsupervised approach to label Type-IV clones and validate them using expert Java programmers. Next, we present a supervised scheme for labeling (or classifying) unknown samples based on labeled samples derived from our first scheme. We evaluate the performance of our schemes using six well-known Java code clone corpora and report on the quality of produced clones in terms of kappa agreement, mean error and accuracy scores. Results show that both schemes produce high quality code clones facilitating future use of machine learning in detecting clones of Type-IV.

Index Terms—Machine Learning, Code Clones, Semantic Clones, AST, PDG, Features, Classification.

I. INTRODUCTION

SOFTWARE maintenance is a critical activity in terms of cost and effort. Many studies show that a software system with many code clones is more difficult to maintain compared to a software system with a fewer clones [1]. Code clones are sequences of duplicate code fragments written in a particular programming language. A code fragment is a part of the source code needed to run a program. It can contain functions, begin-end blocks or a sequence of statements. If a code fragment is similar to another code fragment syntactically or semantically, it is called a clone of the other. If there is a relation between two code fragments such that they are analogous or similar to each other, the two are called a clone pair. Usually, there are two groups of clones. The first group refers to clones which are similar in textual content [1], called Syntactic Clones. The other type of clones includes Semantic Clones, where two code fragments are similar functionally and have no or very low syntactic similarity [2]. Accordingly four types of clones have been identified so far in researches: Type-I (Exact clones), Type-II (Renamed), Type-III (Gapped clones), and Type-IV (Semantic clones).

The first three fall under the syntactic clone category, whereas Type-IV encompasses semantic clones. A number of detectors have been designed for clone detection. Machine learning techniques have been shown to be effective in detecting different type of clones. Success of any supervised technique depends on the availability of adequate amounts of labelled samples for training and testing [2]–[5]. A few benchmark datasets, called as oracle are available for the purpose. They are language dependent collections of code fragments written in programming languages like C or Java. The reality is that most datasets lack Type-IV samples. As a consequence, prior work is also limited up to Type-III clone detection only.

In this work, we relabel existing datasets to include Type-IV clones. Our present work is limited to Java based clones. We propose two schemes (unsupervised and supervised) for labeling Type-IV clones along with other three clone types. In order to identify Type-IV clones, we use Java ByteCode Dependency Graphs (BDG) together with source Program Dependency Graphs (PDG) and Abstract Syntax Trees (AST) of the candidate code fragments or blocks.

The rest of the paper is organized as follows. Prior research is discussed in Section II. Section III-A is dedicated to our proposed schemes. We perform experiments and report results in Section IV. Finally, the paper is concluded in Section V.

II. PRIOR RESEARCH

A number of papers for software clone detection using machine learning have been reported. They use different clone corpora, and they are limited only up to Type-III clones due to lack of labeled semantic clones, which are in fact difficult to label.

Bellon et al.’s [6] datasets are widely used as references by others. Their datasets, introduced in 2007, have become the standard references for evaluation of every new clone detector. Bellon’s dataset identifies only Type-I, II and III clones.

Lavoie and Merlo [7] introduced clone detectors for Type-III clones using the Levenshtein metric, without relying on statistical properties. eclipse. Their work only detects Type-III clones.

Kurtz and Le [8] obtained clones from three open source systems, namely Apache, Python and PostgreSQL. They gen-
erated a set of method level semantic code clones with high confidence to assist evaluation of clone detectors.

Murakami et al. [9] extended Bellon’s dataset of clone references by adding information containing locations of gapped lines. Our approach extends Bellon’s and Murakami’s datasets by adding Type-IV clones so that Type-IV clone detectors can be trained and/or tested.

Svajlenko et al. [10] present a big data clone detection benchmark called JDaDataset 2.0, which is a large Java repository containing 25,000 open-source projects from SourceForge and Google Code.

Our supervised scheme is similar to a traditional machine learning framework. We have two phases, training and testing. In training, we use labeled pairs of cloned blocks from a given corpus. All method blocks are detected from the given corpus using lexical and syntactic analysis. We extract method blocks and perform various preprocessing steps, including trimming

Figure 1: Integrated Scheme for labeling unlabeled clone corpus.

III. LABELING SCHEME FOR CODE CLONE CORPUS

We first propose an unsupervised scheme for labeling clones and validate them with the help of expert programmers. Human expert based labeling of clone types is effective especially in case of semantic clone or Type-IV clone. This is because finding semantic similarity between a pair of code segments is challenging by a machine due to inadequate capability of cognitive reasoning. On the other hand, when the numbers of clone samples are large in any corpus, it is difficult to label them manually by any expert. An automated labeling method in this case is very effective. Hence, as a second attempt we propose a supervised labeling scheme for automated labeling without expert interventions. An integrated flow of our labeling schemes are shown in Figure 1. We next discuss two labeling schemes.

A. Scheme-I: Unsupervised Approach for Relabeling Code Clones

For clone relabeling, we use already labeled “incomplete” clone corpora and relabel them for all types of clones as all but one lack semantic clone labels. We restrict our current study only to Java code. We use two schemes, one based on similarity scores of the target code blocks and the other a supervised approach using an ensemble of classifiers. In this section we discuss our unsupervised scheme based on code block similarity followed by expert validation of the labeling. In both schemes, we use Java source code-based and ByteCode-based features for comparing and labeling. Java ByteCode is a sequence of instructions for the Virtual Machine (JVM) to execute basic functionalities such as conditions and loops [11]. Each ByteCode contains one or more opcodes. ByteCode is tentatively between Java source code and actual machine code. The Java Virtual Machine (JVM) takes the ByteCode and converts it into machine code. When the Java virtual machine (JVM) loads a class file, it is executed by an interpreter. This file contains a stream of ByteCode instructions for each method in the class. ByteCode is a low-level representation for Java programs and hence is likely to be effective for representing the semantics of a program.

As a first step, we tentatively label a clone dataset automatically using Java ByteCode, representing the unified structure of the code. Such clone pairs are not always detectable at source code-level, and we have found that Levenshtein similarity metric [12], for measuring the difference between two sequences, helps identify potential Type-IV clones automatically when ByteCode is used.

The proposed method consists of the following steps. Figure 2 illustrates the workflow of our approach. To explain our steps, we use the two method code blocks given in Figure 2 step 1 as running example.

1) Analyze lexically, normalize and detect method blocks. The first step is to transform and normalize all source files into special token sequences to identify not only identical clones but also similar ones. Figure 2 step 2 gives the two method blocks after lexical analysis and normalization.

2) Identify similar method blocks using Levenshtein distance. This helps identify similar method code blocks using Levenshtein distance after identifying all normalized code blocks.

3) Convert, filter and remove noise method blocks in Java ByteCode classes. This step is to convert all method code blocks into Java ByteCode to identify potential Type-IV (semantic) clones using the Javac compiler. Figure 2 step 3 gives the two Java ByteCode classes after compiling the two method code blocks.

4) Identify similar Java ByteCode classes using Levenshtein metric. This helps identify similar Java ByteCode classes using the Levenshtein metric after filtering and removing all noise. It generates candidate Type-III and Type-IV clones, which may not have been identified at source code level. We choose 50% as a threshold for syntactical similarity.

5) Inspect Manually. This step is a manual inspection of all pairs of methods blocks detected as candidate clones to determine that they are actual clones and their types.

6) Store detected Type-IV clones as references clones. All of the similar codes that are identified in Steps 1, 2, 3 and 4 are stored in the reference clones dataset.

Next, we present a supervised scheme for labeling unknown clone pairs using classification based on high as well as low level code features.

B. Scheme-II: Relabeling Code Clones using Supervised Framework

Our supervised scheme is similar to a traditional machine learning framework. We have two phases, training and testing. In training, we use labeled pairs of cloned blocks from a given corpus. All method blocks are detected from the given corpus using lexical and syntactic analysis. We extract method blocks and perform various preprocessing steps, including trimming
and normalization. We use Abstract Syntax Tree (AST) and Program Dependency Graph (PDG) of the candidate methods along with ByteCodes Dependency Graph (BDG) of the methods as a feature for machine learning.

1) **ByteCode Dependency Graph (BDG):** We attempt to represent the meaning of a program by extracting interdependency relationships among different ByteCode constructs. We represent such dependencies as a graph called ByteCode Dependency Graph (BDG). BDGs represent both data and control dependencies for each operation in the ByteCode. We create a BDG from the ByteCode and extract features from the graph. The BDG features are considered semantic or meaning features. We extract control dependency features by reading the .class file sequentially and by tracking all the instructions that may cause conditional or unconditional branching of the control flow of the code sequence. We consider three types of control instructions: 1) Unconditional branch instructions, 2) Conditional branch instructions, and 3) Compound conditional branch instructions.

2) **Abstract Syntax Tree (AST) and Program Dependency Graph (PDG):** We extract semantic as well as syntactic features directly from the high level source code. The combination of both AST and PDG features may represent the semantics and syntax more accurately, hence, helping in better matching of two target programs. These features are the same as our previous work in [2].

We generate ASTs, PDGs, ByteCode, and BDGs of the blocks and extract features from them. We create a complete feature vector for each block by combining traditional, ASTs, ByteCode, PDGs, and BDGs features. We fuse feature vectors of two target blocks by using Equation 1. 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 types 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 3 demonstrates the workflow of our approach. The following describes the steps.

1) **Build Pairwise Method Blocks.** This step is to pair each method with another method block. This helps compare all blocks in pairs to judge whether or not the two method blocks are identical using a classification algorithm.

2) **Extract Features for Each Two Method Blocks.** In this step, features are extracted from each code fragment using the Java Development Tool (JDT). These features improve the accuracy of Type-III and Type-IV clone detection.

3) **Represent a Pair of Instances as One Vector.** In this step, we create pairs of instances from the original data. Two original instances are represented by feature vectors $X=(x_1, x_2, x_3, ..., x_n)$ and $Y=(y_1, y_2, y_3, ..., y_n)$.
We make a pair instance $Z=(X,Y)$ and represent it as a vector using the following all formula:

$$Z = (|x_1 - y_1|, |x_2 - y_2|, |x_3 - y_3|, \ldots, |x_n - y_m|).$$  

(1)

We perform normalization for Equation 1 by dividing the greatest value of each feature so that all feature values are between 0 and 1.

4) **Labeling Using Classification Model.** After feature extraction, the pair instance is represented as a vector using Equation 1. We feed this data to a classifier. The class label inferred by the classifier will be treated as a label for new clone type for the unlabeled samples.

IV. EXPERIMENTAL EVALUATION

A. Datasets

We consider four clone corpora to use for relabelling using Scheme-I and remaining two additional corpora to be labeled using Scheme-II (see Table I). The corpora are partially labeled. The relabelled clone corpora are available on request.

B. Classification Models Used for Scheme-II

We train and test our model with the decision ensemble of five different commonly used classifiers based on majority voting [16]. We include classification decision from Instance Based Learner (IBK) [17], Bagging [18], Random Committee [19], Rotation Forest [20], and Random Forest [21] classifiers and ensemble them based on majority decisions to obtain the final class label.

C. Experimental Results

We label first four datasets using Scheme-I with the help of expert Java programmers. We choose first four datasets because of their small sizes, convenient for the experts to validate the labeling by our scheme. A brief summary of the extended datasets is given in Table I. The number of paired-blocks we extracted along with numbers of clones of different types for each datasets are reported. We compute Kappa statistic [22] between every pair of observers’ decisions using Equation 2 to validate the labeling. It refers to the probability of agreement among observers or raters.

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

(2)

where, $p_o$ is the relative observed agreement among raters and $p_e$ is the hypothetical probability of chance agreement. Results show that experts are agreed with the labeling for four datasets with the range 60% to 76%.

Since it is difficult to Scheme-I for large datasets, we apply Scheme-II to label them. To generate training samples we take the help of Scheme-I, also validation by the experts. We extracts a total of 2200 code pairs selecting a few samples from both large datasets eclipse-jdtcore and j2sdk1.4.0-javax-swing. The details about the training samples along with characteristics of two large code clone corpora are reported in Table II. We use an ensemble of all the classifiers’ decisions based on majority voting and compute the performance of labeling with respect to Mean Absolute Error (MAE) and Kappa Statistic Agreement (KSA) and report the results in Table II. Models of the classifiers are produced and tested using cross-validation with 10 folds, using Weka and R, 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.

We also use the small datasets for labeling with Scheme-II using the ensemble classifier. Assessment of labeling quality by the ensemble classifier for both small and large datasets are presented in Figures 4 and 5, respectively. The quality for labeling of small datasets with the help of different classifiers are reported in Table III. From the results we observe that the ensemble approach performing better in terms of MAE. However in terms of KSA and Accuracy, all the classifiers show similar performance while labeling.

This experiment shows that the accuracy of a learned code clone model can be improved by using a small number of labeled training instances with a large pool of unlabeled instances. To obtain training labels is expensive, while large quantities of unlabeled paired code clones instances are available. We introduce an approach to learning from labeled and unlabeled paired code clone instances using an ensemble classifier. The approach trains the ensemble classifier using...
available labeled paired code clone instances, and labels the unlabeled paired code clones instances as shown in Figure 3.

Table II shows the number of paired code clones and number of each type of clone, kappa statistic agreement between our model and our experts, and the mean absolute error of the model. The highest kappa statistic agreement is 99% and lowest mean absolute error is 0.07%.

V. Conclusion

In this work, we exploit the benefit of Java ByteCode, to label pair code blocks which might not be syntactically similar at source level but are in fact semantically similar, for generating Type-IV (semantic) clones. We introduce a new dataset of clone references, which contain correct Type-IV clones identified using two schemes for relabeling all types of clones. We use an unsupervised approach for labeling clones and validate them using expert Java programmers and then use a supervised approach for labeling unknown samples based on the known samples derived from the first scheme. As a part of our future endeavor, we would like to extend our work to achieve further improvements to add locational information of gapped lines to label clone types.

REFERENCES