IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 18, 2020, accepted June 16, 2020, date of publication July 1, 2020, date of current version July 20, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3006178

An Automatic Advisor for Refactoring Software Clones Based on Machine Learning

ABDULLAH M. SHENEAMER
Department of Computer Science, Faculty of Computer Science and Information Technology, Jazan University, Jazan 45142, Saudi Arabia
e-mail: asheneamer@jazanu.edu.sa

This work was supported by Jazan University.

ABSTRACT To assist developers refactored code and to enable improvements to software quality when numbers of clones are found in software programs, we require an approach to advise developers on what a clone needs to refactor and what type of refactoring is needed. This paper suggests a unique learning method that automatically extracts features from the detected code clones and trains models to advise developers on what type needs to be refactored. Our approach differs from others, which specifies types of refactored clones as classes and creates a model for detecting the types of refactored clones and the clones which are anonymous. We introduce a new method by which to convert refactoring clone type outliers into *Unknown clone set* to improve classification results. We present an extensive comparative study and an evaluation of the efficacy of our suggested idea by using state-of-the-art classification models

INDEX TERMS Refactoring clone, machine learning, outlier detection, classification, AST and PDG features.

I. INTRODUCTION

Code clones are pairs of code fragments which have a high degree of similarity or which are identical. Code clones might cause software maintenance to be more difficult and a system's source codes more difficult to understand. Code cloning is a popular practice in the software development process for a number of reasons, such as reusing code by ''copy-and-paste'' to increasing the speed of writing the code [1]. There are various clone detector techniques which attempt to find code fragments which have a high number of similarities in the system's source code. Additionally, there have been various refactoring clone tools developed which change the structure of detected code clones without altering code fragment behavior. The refactoring code clones are a method by which to minimize the chances of introducing a bug [2]. Refactoring, or removing, is utilized for improving software comprehensibility and maintainability [3]. Although, Kim *et al.* [4] have shown that clone refactoring cannot solve software quality improvements for two reasons. Firstly, clones often have a short lifespan. Refactoring is less effective if there are block branches in a short distance. Secondly, longer-living clones which have been altered with another element in the same class are difficult to remove or refactor. Additionally, it is a bug which can be simply corrected as the source code can be easily understood, which allows improvement of malleability resulting in code extensibility.

Our approach provides different types of refactoring recommendation to a developer for preventing to remove the positive side of code clones and builds a training model after removing outliers to improve the results. Our tool can be built and used to minimize bugs in a system. Our study can improve clone maintenance by removing duplication code by identifying refactoring clones. Also, the possibility of bad design for a system, difficulty in a system improvement or modification, introducing a new bug, can be decreased by identifying and refactoring clones. In addition, our study can be utilized by various applications such as source code or text plagiarism, malware detection, obfuscated code detection.

In summary, the main contributions of this paper are:

- A presentation of a new machine learning framework that automatically extracts features from the detected code clones and trains models to advise developers on the types of refactored clone code and those which are not refactored.
- We explore a new method by which to convert clone type outliers into an Unknown clone set from the training categories, significantly improving the classification results.
- We present an extensive comparative study and an evaluation of the efficacy of our suggested idea by using state-of-the-art classification models.

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

We have organized the paper thus: Section II introduces the definitions of the cloned and refactored clones. Previous research in this arena is highlighted in Section III. In Section IV, we suggest a new automatic advisor of the refactoring clones approach. In SectionV, we perform an evaluation and comparison of our suggested method and detail the results. Threats to validity are reported in Section VII. Last, our conclusion is in Section VIII.

II. DEFINITIONS OF CODE CLONES AND REFACTORINGS

Developers may apply clone refactorings after detecting code clones to reduce code duplication and minimize the size of the code and the bugs, should there be any bugs in the code. However, some existing works have stated [5] [4] [6] that not all clones need to be removed or refactored. Code clone detection tools can report many code clones in a system so, it would not be easy for developers to locate all detected code clones that need to be refactored [4]. A technique based on machine learning that automatically advises the necessary clones for refactoring would be beneficial.

- *Definition 1 (Code Fragment):* Code fragments are sequences of statements and a range surrounded by *{*and*}*.
- *Definition 2 (Code Clones):* A code fragment is a copy of another, either syntactically or semantically.
- *Definition 3 (Type-I: Exact Clones):* Code fragments are identical, irrespective of their comments and white-spaces, blanks and comments.
- *Definition 4 (Type-II: Renamed Clones):* A code fragment which is syntactically similar except for names of variables, identifiers, types, literals, layouts, white-spaces, blanks, and comments.
  These are created from Type-I clones.
- *Definition 5 (Type-III: Gapped Clones):* Code fragments are sets of type-I and type-II clones, separated by lines that are not syntactically identical, such as addition, deletion, or modification of statements.
- *Definition 6 (Type-IV: Semantic Clones):* A semantic code fragment which performs the same tasks, but which is are implemented in a different manner.
- *Definition 7 (Extract Method (EM)):* is an extracted code fragment as a new method, and which is replacing that code fragment by calling to the new method. The extract method is split into pieces.
- *Definition 8 (Pull-Up Method (PM)):* is a removed similar method found in many classes by the introduction of a generalized method in their common superclass.
- *Definition 9 (Move Method (MM):* is for creating a new method in the class that most frequently employs the method, and which then moves code from the older method into there.

Machine learning and converting clone type of outliers into an Unknown clone set is a good method for building such models. Existing work [2] has shown that machine learning integrates factors which could result in good accuracy in the recommendation of clones. Yet, their work automatically extracted only groups of refactored and non-refactored clones from software repositories and trains the model to implicate clones for refactoring. Our work, meanwhile, extracts different types of refactoring patterns and builds a training model after removing outliers to improve the results.

III. PRIOR RESEARCH

Several studies are related to code clone refactoring. Higo *et al*. [7] suggest a method that refactors code clones using existing refactoring patterns such as the Extract and Pull Up Method. This research performed fully automated refactoring without developer intervention. The developer should evaluate refactoring based on their preference and indicate any clone which is a probable candidate for refactoring. Conversely, our work extracts features and relies on machine learning to build our model and classify clones according to the type of refactored clone and those which are not refactored. Next, the developer evaluated the refactoring clones. Higo *et al*. [8] suggested a method that detects refactoring-oriented code clone to improve the usefulness and applicability of the software maintenance method. Higo *et al*. [9] proposed a refactoring method for merging software clones. Their technique can detect a refactoring-oriented code clone in a general clone detected by token-based or text-based clone detection tools. We refactor clones using AST-based and PDG-based clone detection tools.

Zibran and Roy [10] suggested a model for clone refactoring in object-oriented and procedural source code. Additionally, they proposed a constraint programming (CP) approach for optimal conflict-aware scheduling of code clone refactoring. However, their approach to refactoring the effort model indicates that there are fine grained computations, which could not be manually performed by the developers.

Hotta *et al*. [11] presented a method to refactor Type-III clones by using the Template Method design pattern [19] and program dependency graphs (PDGs). Their tool can only be used on software systems which use object-oriented programming language (OOP). These are semi-automated refactoring clones which exist in the sub-classes.

Tairas and Gray [12] expanded upon Eclipse refactoring to allow additional features amongst cloned code fragments, for instance variations in field accesses, string literals, and method calls without argument. Their tool is an Eclipse plug-in termed CeDAR (Clone Detection, Analysis, and Refactoring) which closes the gaps between refactoring and clone detection. CeDAR only refactors Type-I and Type-II code clones, while our tool is able to detect and refactor Type-I, Type-II, and Type-III.

Saha *et al*. [20] conducted exploratory research on Type-I, Type-II, and Type-III clone evolution in six open source software systems which were written in two programming languages and compared the results with past research to increase understanding of Type-III clone evolution.

Mandal *et al*. [21] investigated the identification of code clones which are crucial for refactoring. They automatically analyzed the history of clone evolution from thousands of software system commit operations downloaded from online SVN repositories, and found specific clone change patterns, Similarity Preserving Change Pattern, whereby code clones which have evolved following this pattern have importance in refactoring.

Mondal *et al*. [22] performed an investigation of the cross-boundary evolutionary coupling of SPCP clones and discovered that SPCP clones have couplings which should not be removed via refactoring. These SPCP clones (i.e., the cross-boundary SPCP clones) need to be considered for tracking together with their relationships across the class boundaries. The non-cross-boundary SPCP clones are important for refactoring.

Wang and Godfrey [6] suggested an automated approach for recommending clones for refactoring by extracting 15 features from refactoring history and training decision tree-based classifiers. Using their datasets this is compared with our approach. We performed a comparison of Wang *et al.*'s approach with our approach that uses Wang *et al.*'s features, which significantly improved the classification results.

Tsantalis *et al*. [13] proposed a method which uses code clones as inputs, and analyzes and examines clones in regard to whether they can be refactored without altering program behaviors. This refactorability analysis tool is used to perform empirical studies on any clone detected by 4 common clone detectors in 9 open-source real systems. However, it is difficult to make all four clone detectors with the same settings since each clone detector has a different type of representation such as CCFinder and Deckard converting code fragment lines into tokens, CloneDR converts the source code lines into AST nodes, and Nicad converts the code fragments into lines. Therefore, each of the tools uses a different notation of similarity and lead to different results.

Mondal *et al*. [14] presented a tool named SPCP-Miner which can automatically identify SPCP clones by examining the software system's clone evolution history. While our approach provides different types of refactoring recommendations to developers to prevent removing the positive side of code clones and builds a training model after removing outliers to improve the results.

Meng *et al*. [15] suggested a method for automated clone refactoring on the basis of systematic edits. They designed and implemented a tool termed RASE. RASE is an automated refactoring tool consisting of four different kinds of clone refactoring methods: using extract methods, using parameterized types, using form templates, and adding parameters. This fully automates the process; therefore, the positive side of code clones would possibly be removed. Mazinanian *et al.* [16] presented a tool for helping developers for refactoring nontrivial clones in Java projects.

Yue *et al*. [2] introduced a tool called CREC, a learning-based method which recommends clones via extraction of 34 features of refactored clones and not refactored from the current or previous software projects. We compare our work for its ability to recommend refactoring clones and show our novel technique performs much better than the CREC technique.

Yoshida *et al*. [17] presented a proactive clone recommendation system for "Extract Method" refactoring using differential node identification. While our approach suggests developers use more than one refactoring type.

Baars and Oprescu [18] extended the Eclipse refactoring to enable more features among cloned code fragments, for example variations in field accesses, string literals, and method calls without argument.

Mondal *et al*. [23] presented a survey on the existing tracking and refactoring tools and identified future research opportunities in refactoring clones. They also compared the state-of-art tools based on quality assessments features. They stated that the automatic refactoring is not able to eliminate the requirement of manual efforts regarding finding refactoring opportunities, post refactoring takes extra effort and time from the quality assurance engineers.

A summary of some of the approaches is reported in Table 1.

IV. PROPOSED APPROACH

In this paper for the *Unknown* set classification, our adopted work model combines supervised learning classifiers and outlier detection for unknown classes. Figures 1 and 2 show diagrams of our framework model.

This paper discusses the common and recent classification algorithms used for refactoring code clone classification and an outlier detection model combined for classifying the test examples as belonging to known or unknown class sizes. The improved performances of our classifier model is reliant upon its closed set validation. Model validation in machine learning is the process whereby trained models are evaluated with testing datasets. The testing dataset in closed set validation contains examples which belong to known classes.

We ran an outlier algorithm for datasets to find the data points which have considerably dissimilarity or inconsistency with the other given data points. Then, the data point classes are changed into unknown classes. After detecting outlier data points, we build our model for closed-set classification and perform analysis of their performance after training. We train and test our classifier with vectors of datasets.

A. AUTOMATIC ADVISOR OF REFACTORING CLONES FRAMEWORK

1) LOCAL OUTLIER FACTOR (DENSITY-BASED APPROACH)

The local outlier factor (LOF) is a score which is calculated by an unsupervised density-based algorithm suggested by Mitchell *et al.* [24] which is reliant on nearest neighbor search and indicates the likelihood that a particular data point is an outlier/anomaly which is ($LOF \approx 1 \Rightarrow No\ Outlier$) and ($LOF \geq 1 \Rightarrow Outlier$)

TABLE 1. Brief summary of refactoring code clones techniques.

Tools/Algorithms	Extract Methods	Pull-up Methods	Move Methods	Other Refactoring Activities	Approach	Language
Higo *et al.* [7]	✓	✓	×	✓	CCFinder Clone detector	Java
Higo *et al.* [8]	✓	✓	×	×	Metric Graph	
Higo *et al.* [9]	✓	✓	✓	✓	Token-based or text-based clone detection tools	Java
Zibran and Roy [10]	✓	✓	×	✓	Constraint programming (CP) technique	Java
Hotta *et al.* [11]	×	×	×	✓	Template Method design pattern and PDGs	Java
Tairas and Gray [13]	✓	✓	×	✓	Eclipse plug-in called CeDAR (Clone Detection, Analysis, and Refactoring)	Java
Wang and Godfrey [6]	✓	✓	×	×	Extracting history features of refactoring and supervised learning classifiers	Java
Tsantalis *et al.* [17]	✓	✓	×	✓	Divide & Conquer and program dependency graph mapping	Java & SQL
Mondal *et al.* [18]	-	-	-	✓	SPCP-Miner for tracking candidates clones	Java
Meng *et al.* [19]	✓	×	×	✓	Abstract template and identifier mappings	Java
Mazinanian *et al.* [20]	✓	✓	×	✓	Lambda expressions	Java
Yue *et al.* [2]	✓	×	×	×	Extracting features of current status and past history and supervised learning classifiers	Java
Yoshida *et al.* [21]	✓	×	×	×	Keystroke tracking & diff analysis and Abstract syntax trees	Java
Baars and Oprescu [22]	×	✓	×	✓	Abstract syntax parser	Java

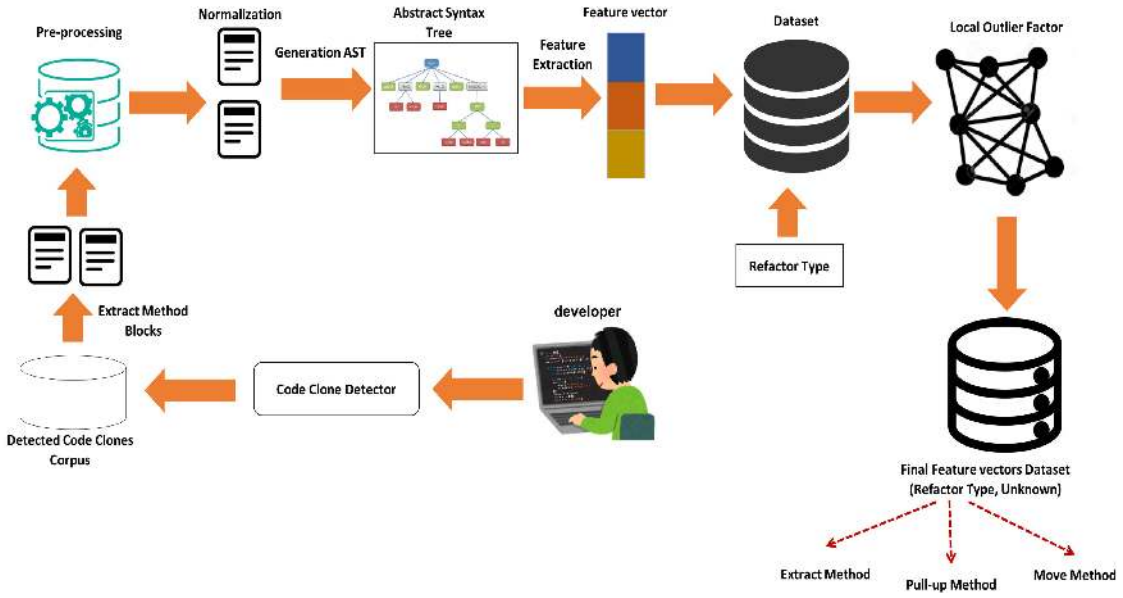


FIGURE 1. Workflow of the proposed outlier framework.

The outlier factor is the anomaly score of each sample. It detects outliers about their neighbors using k-nearest neighbors for estimating the local density. It detects lower density than their neighbors and considers outliers and changes to unknown classes to the dataset. Our framework is a step-by-step approach, firstly, we normalized all source code fragments into special token sequences. Secondly, source code fragments are transformed into abstract syntax trees by existing lexical and syntax analysis tools to detect every block from the provided source files. Thirdly, features are extracted from each of the code fragments using the Java Development Tool (JDT). Then, we created pairs of instances by features vectors from the original data. We feed feature vectors of two target blocks to feature vectors dataset and pass it through the local outlier factor algorithm for outlier detection. Figure 1 demonstrates the workflow of our outlier method.

TABLE 2. Brief description of publicly available Java code clone corpus contained different types of refactoring.

Dataset	# Paired Codes	# Move/Pull Up Method	# Extract method
netbeans	200	54	146
eclipse-jdtcore	400	200	200
EIRC	426	32	394
J2sdk1.4.0-javax	482	200	282
eclipse-ant	522	262	260
cocoon	655	141	514

2) CLASSIFICATION MODEL

Machine learning is an understanding of the basic principles of learning through mathematics, statistics, and computer science. In other words, it is a branch of artificial intelligence that aims to build a mathematical model based on data, a well-known training group for making decisions or predictions without being explicitly programmed. Machine learning contains many sub-domains and applications, including statistical learning methods, neural networks, instance-based learning, data mining, image recognition, natural language processing (NLP), and enhanced learning [24].

We adopt machine learning algorithms to build detection models for detecting refactored types. For each classifier, the default configuration and parameters suggested by Weka. Like any other machine learning framework, our framework has two phases, training and testing. In training, we use labeled pairs of refactored code from a given corpus. We train and test our proposed framework using classification models, starting with the popular Bagging [25] and K-nearest neighbors (KNN) [26] model to a recently published class implementation, which can be used simply to solve both classification and regression problems. Forest by Penalizing Attributes (Forest PA) [27] is a decision forest algorithm building sets of very accurate decision trees using the strengths of all non-class attributes in a dataset and simultaneously imposing penalties (disadvantageous weights). Random Forest (RF) [28] creates multiple trees for classification.

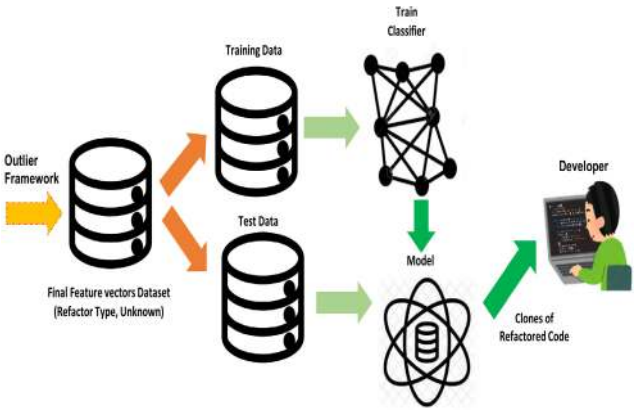


FIGURE 2. Workflow of the proposed refactoring recommendation framework.

In Figure 2, our approach shows the major steps of the automatic refactoring clone recommendation prediction model. Our model uses 10 fold cross-validation to compare the performance of classifiers before and after applying the local outlier algorithm on the datasets and the application of supervised learning classifiers to predict recommendation refactoring clones. The final stage is the decision about whether the class is refactored (Refactoring clone type and Unknown clone type).

TABLE 3. Percentage of clone types can be refactored.

Dataset	Refactoring (Type-I/II)	Refactoring (Type-III)
netbeans	100%	100%
eclipse-jdtcore	99.5%	77.3%
EIRC	85.5%	100%
J2sdk1.4.0-javax	84.85%	96.83%
eclipse-ant	100%	70.8%
cocoon	9.2%	90.2%

V. EXPERIMENTAL EVALUATION

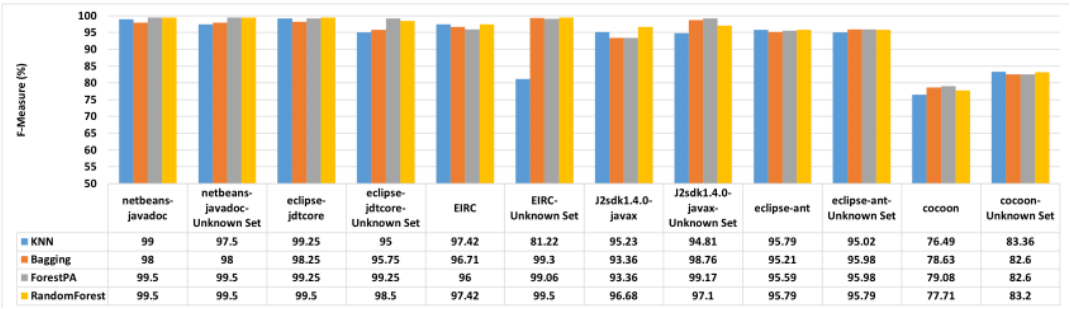
A. DATASETS

We considered six publicly available datasets, which are *netbeans*, *eclipse-jdtcore*, *EIRC*, *J2sdk*1.4.0-*javax*, *eclipse-ant*, and *cocoon*. The details of the datasets are given in Table 2 and Table 3 summarizes the data distribution in six publicly available datasets in terms of clone types. For this experiment, we consider every type of clone in datasets which are 6 lines or 50 tokens or longer, since this is the standard minimum clone size for benchmarking [29] and [30]. We have divided Type-I and Type-II clones into Move and Pull Up Method refactoring and Type-III clones into Extract method refactoring.

We also used six open projects [2]: *Axis*2, *Eclipse.jdt.core*, *ElasticSearch*, *JFreeChart*, *JRuby*, and *Lucene*. These open projects include refactoring and non-refactoring clones. The details of the datasets are given in Table 4. We used these datasets for two reasons. First, these datasets were used by prior research [6] and [2]. Second, we used Yue *et al.*'s datasets to compare our approach with previous approaches [6] and [2]. Yue *et. al.* randomly chose a subset of clone groups from clone sets without refactoring clones to create a balanced data set of positive and negative examples. The number of Non-refactoring clones' subset is equal to that of reported Refactoring clones.

B. EVALUATION

We generated excessive results for assessing the strength of our suggested model in detecting clones and advising refactored clones together with all other types. We experimented with various numbers of features and with different data



	netbeans-javadoc	netbeans-javadoc-Unknown Set	eclipse-jdtcore	eclipse-jdtcore-Unknown Set	EIRC	EIRC-Unknown Set	J2sdk1.4.0-javax	J2sdk1.4.0-javax-Unknown Set	eclipse-ant	eclipse-ant-Unknown Set	cocoon	cocoon-Unknown Set
KNN	99	97.5	99.25	95	97.42	81.22	95.23	94.81	95.79	95.02	76.49	83.36
Bagging	98	98	98.25	95.75	96.71	99.3	93.36	98.76	95.21	95.98	78.63	82.6
ForestPA	99.5	99.5	99.25	99.25	96	99.06	93.36	99.17	95.59	95.98	79.08	82.6
RandomForest	99.5	99.5	99.5	98.5	97.42	99.5	96.68	97.1	95.79	95.79	77.71	83.2

FIGURE 3. Performance of all the candidate classifiers with respect to F-measure assessment metric on different datasets.

TABLE 4. Brief description of publicly available Java code clone corpus contained Refactored Clones and Non-Refactored Clones [2].

Dataset	Total	# Refactored Clones	# Non-Refactored Clones
axis2-java	86	43	43
eclipse-jdt.core	212	106	106
elasticsearch	66	33	33
jfreechart	118	59	59
jruby	130	65	65
lucene	54	27	27

instances to demonstrate that our approach can achieve a high degree of detection accuracy. Because of space limitations, we reported only the best-performing classifiers for most of the experiments and compared them with state-of-the-art automatic recommendation refactored clone methods. To generate AST from a given block to extract features, we used Eclipse Java Development Tools (JDT).

1) PERFORMANCE OF DIFFERENT SUPERVISED LEARNING CLASSIFIERS

We ran four different classifiers *Bagging*, *KNN*, *ForestPA* and *RF* to compare the performance of classifiers before and after applying the local outlier algorithm on the datasets. All the candidate models were tested and trained by using 10 fold cross-validation, whereby we ensured that the ratios between refactoring clones and unknown classes were identical in each of the folds and the same as in the overall dataset. Figure 3 highlights the comparisons of all four classifiers using the outlier algorithm and without using the outlier algorithm respectively on our datasets. The experimental results show that the decision tree algorithms, such as ForestPA and RF, which are ensemble learning, achieved better outcomes among all the classifiers. This is because RF combines a large number of independent trees which are randomly trained.

2) PERFORMANCE COMPARISON WITH RECEIVER OPERATING CHARACTERISTIC CURVE (ROC)

The classes in the classification curves are: *Extract_Method*, *Move/Pull_Method*, *Unknown*. The receiver operating characteristic (ROC) curve analysis is used as the measure of performance. The higher value for the area under the ROC curve indicates that the classifier the better it is at distinguishing refactoring clones in the classes. The Random forest (RF) had a better performance in every class. Therefore, we show an ROC curve for all datasets using RF.

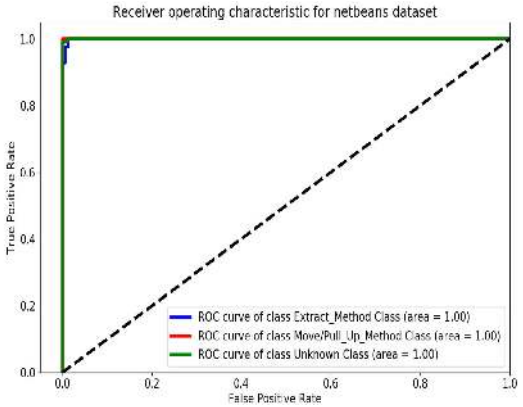


FIGURE 4. Receiver operating characteristic curve (Roc) for netbeans dataset.

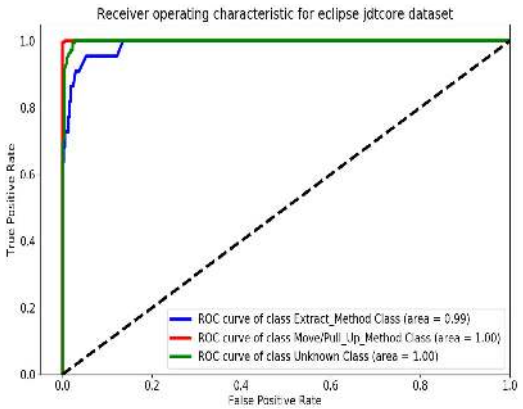


FIGURE 5. Receiver operating characteristic curve (Roc) for eclipse.jdtcore dataset.

RF has a ROC of 1.00 area under all classes on the netbeans dataset as shown in Figure 4. The ROC curve area under *Extract_Method* class is 0.99; other classes in the ROC curve area is 1.00 on *eclipsejdtcore* dataset respectively, as shown in Figure 5. On the *EIRC* dataset, the ROC curve area under *Extract_Method* and *Unknown* class were 0.97; ROC curve area under *Move/Pull_Method* class was 0.92 as shown in Figure 6. We get also an ROC curve area under *Extract_Method* class equal to 0.99 and 0.98 for other classes on *j2sdk*1.4.0-*javax* dataset, as shown in Figure 7. The ROC curve area under *Extract_Method* class is 0.87; *Move/Pull_Method* class ROC curve area is 0.94; the ROC

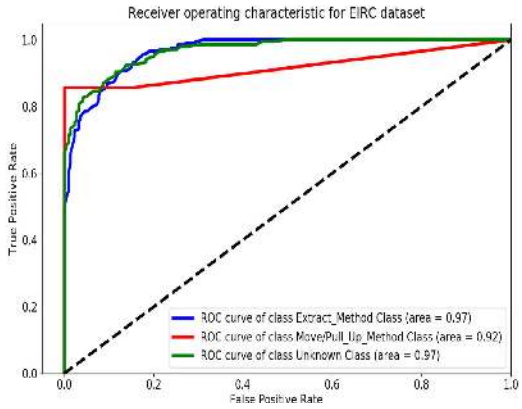


FIGURE 6. Receiver operating characteristic curve (Roc) for EIRC dataset.

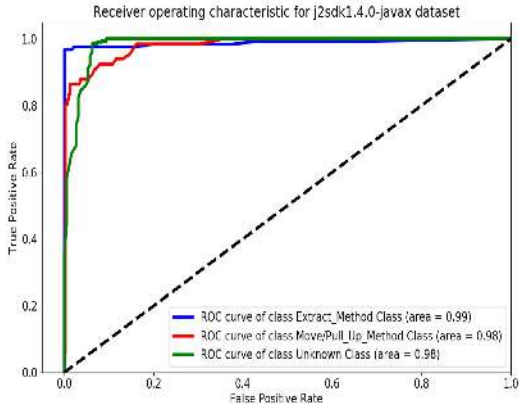


FIGURE 7. Receiver operating characteristic curve (Roc) for j2sdk1.4.0-javax dataset.

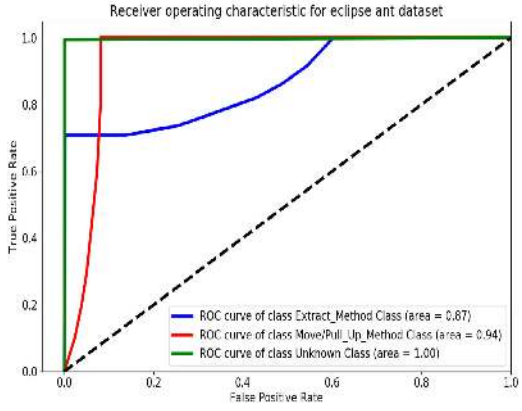


FIGURE 8. Receiver operating characteristic curve (Roc) for eclipse-ant dataset.

curve area under Unknown class is 1.00 on *eclipse-ant* dataset as shown in Figure 8. The ROC curve area under *Extract_Method* and *Move/Pull_Method* classes is 0.90, *Unkonwn* class is 1.00 on cocoon as shown in Figure 9. Because of the imbalance of the dataset, Precision, Recall, F1-measure was utilized for additional evaluation of the classifiers

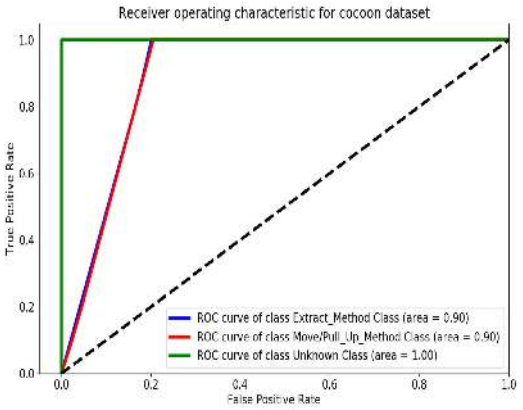


FIGURE 9. Receiver operating characteristic curve (Roc) for cocoon dataset.

3) PERFORMANCE COMPARISON WITH OTHER STATE-OF-THE-ART APPROACHES

However, because of the lack of refactoring methods research papers, we compared how our method performed with the existing and current approaches of refactoring clones, using their reported Precision, Recall and F-score results on datasets. We compare our method using *CREC* features set [2] and the WangWei features set [6] with the *CREC* approach and Wang *et al.* approach. Interestingly, most of the refactoring methods are incapable of recommending the types of refactoring clones. Wang and Godfrey [6] produced a machine learning-based clone recommendation method, which extracted fifteen features for clone relation, the context of clones and cloned code snippet categories. Ten features are representative of the current status of individual clones, one of the features reflects the evaluation history of each individual clone, three features describe the relationships between clones, and one feature shows any syntactic differences between clones. The *CREC* approach [2] extracts five categories of features: two categories model the present version and historic evolution of an individual clone, and three categories reflect the locations, code differences, and co-change relationships amongst clone peers of each of the groups. Figures 10 and 11 show a comparison of our results using the RF algorithm with the state-of-the-art approaches based on different assessment metrics on six publicly available datasets. From the results it can be seen that our approach performs much better based on the recall, precision, F-measure and accuracy metrics. Results clearly show whether our method is more effective in advising to refactor clones, together with a comparison to the other methods. Our approach achieved the highest accuracy, at approximately 93% in *axis2-java* project using *CREC* Features set and approximately 87% in *eclipse.jdt.core* project using Wang Wei features set. However, the method achieved better results based on recall, precision and F-measure metrics using *CREC* Features set, except for the *jfreechart* and *lucene* projects, which were better than our results. Additionally, our method achieved better results based on recall, precision

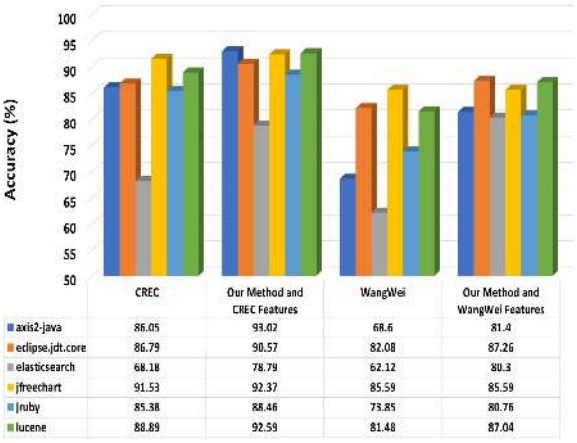
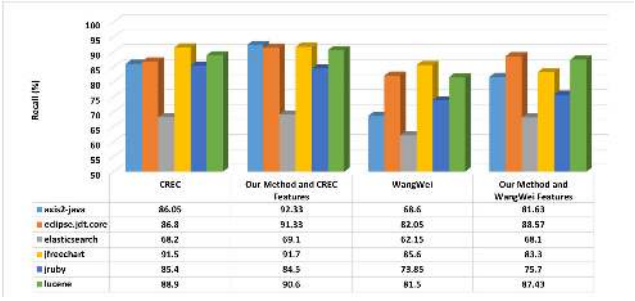
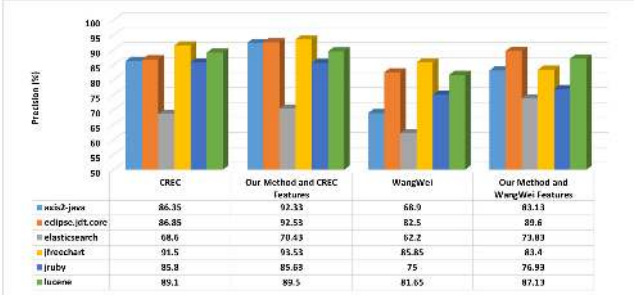


FIGURE 10. Performance comparison of different approaches with respect to accuracy metrics on six publicly available datasets.



(a) *Recall Assessments*



(b) *Precision Assessments*

(c) *F-measure Assessments*

FIGURE 11. Performance comparison of different refactoring methods with respect to different assessment metrics on six publicly available datasets.

and F-measure metrics using the Wang Wei Features set in all cases. Table 5 reports the results of our proposed work compared with CREC and WanWei techniques [2] and [6].

VI. COMPARISON DISCUSSION

There are many methods which automatically remove clones via application of refactoring [13], [15], [16]. Such methods primarily perform an extraction of a method by parameterizing any clone differences and factorizing the common part. Many researchers have carried out a program dependency analysis to bring additional automation to the Extract Method refactorings for near-miss clones. These are clones containing different statements or divergent program syntactic structures. In particular, Meng *et al.* found continually updated clones in history and automated refactorings for removing duplicated codes and to lower repetitive coding efforts [15]. Tsantalis *et al.* automatically detected any difference between each clone, and additionally determined if it was safe to parameterize those differences without altering their program behaviors [13]. Later, Tsantalis *et al.* built an additional tool that used lambda expressions for refactoring clones [1]. As stated in past research [15], the above mentioned research primarily focuses on automated refactoring feasibility rather than desirability.

Contrastingly, our research performs an investigation of refactoring desirability. Through the extraction of various elements which reflect the possible costs and benefit of refactoring, we rely on machine learning for modeling the complex interactions between clone-based features which are already refactored or not refactored by developers. Thus, the trained classifiers can imitate the human mental processes of evaluating refactoring desirability, and also indicate the clones which are likely to be refactored by developers.

To determine how appropriate refactoring is for each of the clones, we utilize the machine learning approach to suggest clones for refactoring by learning from refactoring abstract syntax trees and program dependency of clones. In particular, we collected examples of clone refactoring. Of the six target systems, the extracted features are determined which are related to cloned code. After assemblage of the features from both ''refactored'' and ''unrefactored'' clones, we considered clone refactoring as a classification issue. Therefore, our approach captures the essence of analysis by the developers. In particular, we trained classifiers to learn from features from both refactored (with its types) and unrefactored clone instances which occur in clone evolution. Our method was evaluated by 10-fold cross validation, and cross-project validation. This frees the developers from manually collection and analysis of information for making decisions, which allows developers to focus on the clones which seem the most appropriate for refactoring.

Our approach performs an extraction of clone code which can be refactored using a program dependency graph (PDG) and an abstract syntax tree (AST). Consequently, it can identify re-organized clone codes and can be used for process extraction and structure analysis of the differences of the statement, it can conduct variable identification and extraction. Contrastingly, jDeodrant performs refactoring of differing clones, but if the expressions appear in the differences they are unable to be safely parameterized.

TABLE 5. Results of our proposed work.

Dataset	Tool	Accuracy (%)	Class	F1-Measure	Precision	Recall
axis2-java	CREC	86.05	Refactored Clones	0.867	0.830	0.907
			Non-Refactored Clones	0.854	0.897	0.814
	Our Method with CREC Features	93.02	Refactored Clones	0.880	0.880	0.880
			Non-Refactored Clones	0.917	0.917	0.917
			Unknown Clones	0.973	0.973	0.973
	WangWei Features	68.60	Refactored Clones	0.703	0.667	0.744
			Non-Refactored Clones	0.667	0.711	0.628
	Our Method with WangWei Features	81.40	Refactored Clones	0.771	0.711	0.844
			Non-Refactored Clones	0.706	0.783	0.643
			Unknown Clones	0.980	1.000	0.962
eclipse.jdt.core	CREC	86.79	Refactored Clones	0.870	0.855	0.887
			Non-Refactored Clones	0.865	0.882	0.849
	Our Method with CREC Features	90.57	Refactored Clones	0.893	0.878	0.908
			Non-Refactored Clones	0.893	0.898	0.888
			Unknown Clones	0.971	1.000	0.944
	WangWei Features	82.08	Refactored Clones	0.830	0.788	0.877
			Non-Refactored Clones	0.810	0.862	0.764
	Our Method with WangWei Features	87.26	Refactored Clones	0.859	0.806	0.919
			Non-Refactored Clones	0.816	0.882	0.759
			Unknown Clones	0.989	1.000	0.979
elasticsearch	CREC	68.18	Refactored Clones	0.704	0.658	0.758
			Non-Refactored Clones	0.656	0.714	0.606
	Our Method with CREC Features	78.79	Refactored Clones	0.500	0.545	0.462
			Non-Refactored Clones	0.629	0.647	0.611
			Unknown Clones	0.959	0.921	1.000
	WangWei Features	62.12	Refactored Clones	0.638	0.611	0.667
			Non-Refactored Clones	0.603	0.633	0.576
	Our Method with WangWei Features	75.76	Refactored Clones	0.843	0.768	0.935
			Non-Refactored Clones	0.467	0.700	0.350
			Unknown Clones	0.729	0.747	0.758
jfreechart	CREC	91.53	Refactored Clones	0.915	0.915	0.915
			Non-Refactored Clones	0.915	0.915	0.915
	Our Method with CREC Features	92.37	Refactored Clones	0.929	0.897	0.963
			Non-Refactored Clones	0.882	0.909	0.857
			Unknown Clones	0.964	1.000	0.931
	WangWei Features	85.59	Refactored Clones	0.850	0.889	0.814
			Non-Refactored Clones	0.862	0.828	0.898
	Our Method with WangWei Features	85.59	Refactored Clones	0.779	0.769	0.789
			Non-Refactored Clones	0.721	0.733	0.710
			Unknown Clones	1.000	1.000	1.000
jruby	CREC	85.38	Refactored Clones	0.861	0.819	0.908
			Non-Refactored Clones	0.846	0.897	0.800
	Our Method with CREC Features	88.46	Refactored Clones	0.806	0.744	0.879
			Non-Refactored Clones	0.737	0.840	0.656
			Unknown Clones	0.992	0.985	1.000
	WangWei Features	73.85	Refactored Clones	0.764	0.696	0.846
			Non-Refactored Clones	0.707	0.804	0.631
	Our Method with WangWei Features	80.77	Refactored Clones	0.783	0.726	0.849
			Non-Refactored Clones	0.684	0.771	0.614
			Unknown Clones	0.804	0.811	0.808
lucene	CREC	88.89	Refactored Clones	0.893	0.862	0.926
			Non-Refactored Clones	0.885	0.920	0.852
	Our Method with CREC Features	92.59	Refactored Clones	0.806	0.744	0.879
			Non-Refactored Clones	0.914	0.941	0.889
			Unknown Clones	0.974	1.000	0.950
	WangWei Features	81.48	Refactored Clones	0.821	0.793	0.852
			Non-Refactored Clones	0.808	0.840	0.778
	Our Method with WangWei Features	87.03	Refactored Clones	0.857	0.882	0.833
			Non-Refactored Clones	0.857	0.857	0.857
			Unknown Clones	0.903	0.875	0.933

Not every difference requires parameterization. Additionally, refactoring cannot cover every refactoring situation.

Regarding the clone code refactoring recognition standard, which clone code requires refactoring and which clone code requires keeping track of attention, no established standard currently exists. JDeodorant only evaluates the predictions by collection of the attribute metrics of some clone codes and does not consider the actual changes of the software code. Even though these methods can evaluate if the clone code requires refactoring, because of the lack of qualitative analysis for evolution processes and the actual nature of the code cloning operations, the maintenance personnel cannot accurately gain the information needed.

Tsantalis *et al.* automatically detected any difference between clones and performed an assessment of if it was safe to parameterize those differences without altering program behavior. Our belief is that our approach uses AST and PDG to lower the ambiguity of statement matching the approach of Tsntalis *et al.* Yet, Tsntalis et al's approach used full automation, which might remove the positive side of the code clones. Whilst our approach uses machine learning it does offer a good method by which to construct such a model and may prevent removal of the positive side of code clones. Previous work [6] shows that the use of machine learning is for integrating factors which may result in more accurate recommendation of clones. Our model utilizes the outlier/anomaly algorithm which can be included into the model of open-set classification as unknown classes to discover the data points which have less similarity or inconsistency with the other given data points. Next, these data point classes are transformed into unknown classes. After detection of the outlier data points, we construct out model for closed-set classification and analyze their performance after training.

Tsantalis *et al.* [31] suggest using lambda expressions (anonymous functions in Java 8) for addressing statement/block level differences in cloned methods. In this instance different lambda expressions are used as parameters to a merged method and substitute various statements. The same nine Java systems [23] were used for collecting clones. Firstly, the authors used their approach to refactor 12,602 CPs (covered by unit tests) from ''JFreeChart'', assessed as refactorable. Furthermore, they assessed 18,402 Type III CPs from nine subject systems and approximately 60% of these were reported as refactorable (but not actually refactored or tested).

Our approach used AST and PDG matching algorithms by extending the ASTMatcher superclass given in the Eclipse JDT framework. Our implementation also returns a differences list found amongst the matched PDG nodes used in examining the preconditions.

A number of research types indicate that refactoring improves software maintainability. It is required to recommend a clone code which requires refactoring to maintenance personnel. However, the management and maintenance of clones has gained extensive attention recently, the methods for predicting clone code refactoring are continuously proposed, yet the existing tools for clone code refactoring is imperfect, the degree of automation is low, and some errors are introduced into the software. The technology required for clone code refactoring remains unexplored, therefore it is difficult to disseminate throughout the industry; no effective tools exist for the recommendation and management of clones. Thus, exploring intelligent recommendation or advisor methods is a vitally important content of clone code refactoring research.

VII. THREATS TO VALIDITY

The datasets are restricted to Java-based clones. We performed an evaluation and comparison of our method and *CREC* and *WangWei* techniques, with only Refactoring and Non-Refactoring clones extracted from their six open source projects with a lack of labels of the refactoring types. The evaluation may have a huge impact on the results if we compare our method with current methods using refactoring type clones.

VIII. CONCLUSION

This paper suggests a learning method which automatically extracts features from the detected code clones and trains the models to advise the developers in regard to what a clone needs to be refactored and what is its type. We introduce a new method of converting clone type outliers into an Unknown clone to improve classification results. We present an extensive comparative study and perform an evaluation of the efficacy of our suggested idea by using state-of-the- art classification models.

- We present a new machine learning framework that automatically extracts features from the detected code clones and trains models to advise the developers on the type of refactored clone code and those which are not refactored.
- We explore a new method by which to clone types of outliers into an Unknown clone from the training categories, which significantly improves the classification results.
- We present an extensive comparative study and an evaluation of the efficacy of our suggested idea by using state-of-the-art classification models.

We used four classification models to obtain their relative performance. The experimental results suggest that our approach has high value in achieving high automated advising refactored clone accuracy. In future, we would like to increase the scope of work to achieve additional improvements, for example, by using set classification and deep learning.

REFERENCES

[1] Y. Dang, S. Ge, R. Huang, and D. Zhang, ''Code clone detection experience at microsoft,'' in *Proc. 5th Int. Workshop Softw. Clones*, 2011, pp. 63–64.
[2] R. Yue, Z. Gao, N. Meng, Y. Xiong, X. Wang, and J. D. Morgenthaler, ''Automatic clone recommendation for refactoring based on the present and the past,'' in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2018, pp. 115–126.
[3] S. Kodhai and S. Kanmani, ''Method-level code clone modification using refactoring techniques for clone maintenance,'' *Adv. Comput. Int. J.*, vol. 4, no. 2, pp. 7–26, Mar. 2013.
[4] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, ''An empirical study of code clone genealogies,'' *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, 2005, pp. 187–196.
[5] N. Göde and R. Koschke, ''Frequency and risks of changes to clones,'' in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 311–320.
[6] W. Wang and M. W. Godfrey, ''Recommending clones for refactoring using design, context, and history,'' in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, Sep. 2014, pp. 331–340.
[7] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, ''Refactoring support based on code clone analysis,'' in *Proc. 135Int. Conf. Product Focused Softw. Process Improvement*. Cham, Switzerland: Springer, 2004, pp. 220–233.
[8] Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue, and K. Words, ''ARIES: Refactoring support environment based on code clone analysis,'' in *Proc. IASTED Conf. Softw. Eng. Appl.*, 2004, pp. 222–229.

[9] Y. Higo, S. Kusumoto, and K. Inoue, "A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system," *J. Softw. Maintenance Evol. Res. Pract.*, vol. 20, no. 6, pp. 435–461, Nov. 2008.
[10] M. F. Zibran and C. K. Roy, "A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring," in *Proc. IEEE 11th Int. Work. Conf. Source Code Anal. Manipulation*, Sep. 2011, pp. 105–114.
[11] K. Hotta, Y. Higo, and S. Kusumoto, "Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph," in *Proc. 16th Eur. Conf. Softw. Maintenance Reeng.*, Mar. 2012, pp. 53–62.
[12] R. Tairas and J. Gray, "Increasing clone maintenance support by unifying clone detection and refactoring activities," *Inf. Softw. Technol.*, vol. 54, no. 12, pp. 1297–1307, Dec. 2012.
[13] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, "Assessing the refactorability of software clones," *IEEE Trans. Softw. Eng.*, vol. 41, no. 11, pp. 1055–1090, Nov. 2015.
[14] M. Mondal, C. K. Roy, and K. A. Schneider, "SPCP-miner: A tool for mining code clones that are important for refactoring or tracking," in *Proc. IEEE 22nd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, Mar. 2015, pp. 484–488.
[15] L. Hua, M. Kim, and K. S. McKinley, "Does automated refactoring obviate systematic editing?" in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 1, May 2015, pp. 392–402.
[16] D. Mazinanian, N. Tsantalis, R. Stein, and Z. Valenta, "JDeodorant: Clone refactoring," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. Companion (ICSE-C)*, May 2016, pp. 613–616.
[17] N. Yoshida, S. Numata, E. Choiz, and K. Inoue, "Proactive clone recommendation system for extract method refactoring," in *Proc. IEEE/ACM 3rd Int. Workshop Refactoring (IWoR)*, May 2019, pp. 67–70.
[18] S. Baars and A. Oprescu, "Towards automated refactoring of code clones in object-oriented programming languages," EasyChair, Tech. Rep., 2019.
[19] E. Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software*. New Delhi, India: Pearson, 1995.
[20] R. K. Saha, C. K. Roy, K. A. Schneider, and D. E. Perry, "Understanding the evolution of type-3 clones: An exploratory study," in *Proc. 10th Work. Conf. Mining Softw. Repositories (MSR)*, May 2013, pp. 139–148.
[21] M. Mandal, C. K. Roy, and K. A. Schneider, "Automatic ranking of clones for refactoring through mining association rules," in *Proc. Softw. Evol. Week IEEE Conf. Softw. Maintenance, Reeng., Reverse Eng. (CSMR-WCRE)*, Feb. 2014, pp. 114–123.
[22] M. Mondal, C. K. Roy, and K. A. Schneider, "Automatic identification of important clones for refactoring and tracking," in *Proc. IEEE 14th Int. Work. Conf. Source Code Anal. Manipulation*, Sep. 2014, pp. 11–20.
[23] M. Mondal, C. K. Roy, and K. A. Schneider, "A survey on clone refactoring and tracking," *J. Syst. Softw.*, vol. 159, Jan. 2020, Art. no. 110429.
[24] T. M. Mitchell, *Machine Learning. 1997*, vol. 45, no. 37. Burr Ridge, IL, USA: McGraw-Hill, 1997, pp. 870–877.
[25] L. Breiman, "Bagging predictors," *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, Aug. 1996.
[26] D. W. Aha, D. Kibler, and M. K. Albert, "Instance-based learning algorithms," *Mach. Learn.*, vol. 6, no. 1, pp. 37–66, Jan. 1991.
[27] M. N. Adnan and M. Z. Islam, "Forest PA: Constructing a decision forest by penalizing attributes used in previous trees," *Expert Syst. Appl.*, vol. 89, pp. 389–403, Dec. 2017.
[28] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.
[29] 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.
[30] 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*, 2016, pp. 597–600.
[31] N. Tsantalis, D. Mazinanian, and S. Rostami, "Clone refactoring with lambda expressions," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, May 2017, pp. 60–70.

ABDULLAH M. 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 and the former Vice-Dean of the Faculty of Computer Science and Information Technology, Jazan University, Saudi Arabia. His research interests include data mining, machine learning, software engineering, and malware analysis. His current research interests include software clone and refactoring software clone, malware detection, and code obfuscation detection using machine learning approaches. He has published several papers in reputed international journals and conferences. He had reviewed several articles in reputed journals, such as IEEE Access, the IEEE Transactions on Parallel and Distributed Systems, *Information Sciences* (Elsevier), and the *Frontiers of Computer Science*. He is also a Senior Meta Reviewer of the IEEE International Conference on Machine Learning and Applications.