A Refactoring Advisor for Enhanced Cloned Software: Based on Several Machine Learning Techniques

Badri Narayanan K[1*], Sreeja Nukarapu[2*],
Devatha Krishna Sai [3*] And Bharath Reddy Gudibandi[4*]

[1*]School of computer science and engineering (SCOPE), VIT-AP University, Amaravathi, 522237, Andhra Pradesh, India.
[2*]Computer science and engineering, Vallurupalli Nageswara Rao Vignana Jyothi Institute of Engineering Technology College, Secunderabad, 500090, Telangana, India.
[3,4*]School of Electronics and communication engineering (SENCE), VIT-AP University, Amaravathi, 522237, Andhra Pradesh, India.

*Corresponding author(s). E-mail(s): badrinarayanan78@gmail.com ; nukarapusreeja14@gmail.com; krishnasaidevatha@gmail.com; bharathreddy.gudibandi@vitap.ac.in;

Abstract. Software development is time-consuming and frequently monotonous. The risk of code getting copied and pasted increases as the number of developers working on a project rises. When a program contains numerous duplicates, the quality of the code can be raised by giving developers instructions on how to rework the clone and what needs to be refactored. Code that needs to be refactored can be found by looking at clones with an automatic refactoring advisor. The advisors may be incorporated into current IDEs. Here, we describe a cutting-edge learning technique that automatically extracts properties from discovered code clones and trains models to give developers advice on how to eliminate code duplication. We demonstrate that our method outperforms earlier methods in terms of both the accuracy of the provided refactoring recommendations and the ability to automatically extract the proper parameters for performing refactoring on code clones. In contrast to prior approaches' Class-based approach, we created a model to distinguish between refactored and anonymous copies. We demonstrate how the learned model can be used to rate each cloned piece of code in a codebase and evaluate whether it needs to be refactored. We describe a novel method for turning anomalous refactoring clone types into unknown clone set participants, 1 which is a more reliable solution than prior work that employed thresholds of similarity to identify refactoring clones and offers a thorough analysis.

Keywords: Clone, machine learning, outlier detection, categorization, abstract syntax tree (AST)

1 Introduction

A significant portion of software development is dedicated to code refactoring, a process that involves restructuring existing code to enhance its comprehensibility or correctness, often adhering to the principle of least astonishment.

When two programs are functionally identical or nearly so, they are referred to as ”cloned.” In scenarios where program clones are prevalent, such as large code bases shared by

ICISML 2024, January 05-06, Pune, India
Copyright © 2024 EAI
DOI 10.4108/eai.5-1-2024.2342581

multiple developers, refactoring becomes a laborious task. Code clones can impede system analysis, complicate maintenance, and force programmers into a less intuitive and forgiving environment compared to non-clone code. The software development process can be expedited by creating ”clones” of existing code through copying and pasting relevant sections.

Addressing the complex issue of code cloning in software engineering involves the use of various clone detection techniques to identify sections of source code that are almost identical. Refactoring clone tools have emerged, making subtle changes to copied code structure without affecting functionality, thus eliminating redundant code and preserving shared information between programs. While these tools can significantly reduce cloned code, achieving 100 percent elimination remains challenging.

A more effective approach to minimizing cloned code involves detection and elimination at the source-code level. This is accomplished by deleting unnecessary code and improving software readability and maintainability. However, there is a potential risk as refactoring may fail when a branching condition is near a block, and fusing two clones of the same kind produces a hybrid clone that is challenging to eliminate or restructure.

Code cloning often occurs in large application codebases over an extended period with multiple developers, and merging code may compromise quality by introducing unexpected errors. A clean and well-documented codebase facilitates resolution of such issues, supporting a broader range of use cases.

Our approach aims to preserve the benefits of code duplication by offering programmers various rewriting options and refining the results by removing outliers from the training set. This results in an optimal mapping between the original block and its derived variant.

Implemented on a large scale within software development organizations, our method has the potential to reduce software defects. Many software companies already use automated tools to analyze and detect instances of code cloning in their systems. Eliminating unnecessary steps can boost system performance, facilitate future modifications, and minimize the introduction of new issues, thus reducing clone upkeep expenses.

Our findings are anticipated to be useful in various settings, including anti- plagiarism software, virus detection code analysis. The most significant results of this study include the presentation of a novel machine-learning framework for automatically collecting features from detected code clones. We train models to guide developers in differentiating between refactored and un-refactored clone code.

Furthermore, our novel approach enhances classification outcomes by reclassifying clone-type outliers into a collection of Unknown clones chosen from the training sets. This improves the robustness of the classifier by eliminating poor matches and reducing the impact of noise in the training data. We also address the issue of class imbalance caused by duplicate code in existing systems, employing techniques to alleviate the problem and state-of-the-art categorization techniques.

In conclusion, our results demonstrate that the proposed approach can pro- duce accurate and robust classifications across a range of datasets without requiring domain-specific knowledge or human annotations. We have shown that constructing an accurate Clone classification model from a set of machine learning algorithms provides complementary insights into the inner workings of clone systems.

2 Related Work

”Expertise in detecting code clones at Microsoft”: The practice of reusing portions of source code from completed projects is widespread in the software development industry. This practice allows teams to rapidly build systems and facilitates knowledge sharing within the

organization. However, it also introduces challenges for software maintenance.

A larger code base increases the likelihood of creating more copies, making it challenging to maintain clone consistency and eliminate unnecessary clones when developing commercial software at scale. For example, developers may overlook whether different copies of the same class are functioning correctly or may be unaware of the duplication. This white paper will address some common reasons Microsoft programmers have given for locating code duplicates. Additionally, we include feedback from Microsoft developers who have deployed the XIAO code clone detection tool, witnessing substantial efficiency gains.

While clone detection has traditionally been a tool for technical debt remediation and maintenance tasks, previous research has primarily focused on a manual approach to reviewing clones.

”Refactoring suggestions that use historical data and existing code clones automatically”: There’s a significant likelihood that developers would pay equal attention to each copy if they appeared simultaneously. To assist developers in removing clones based on past and present data, such as the co-evolution of clones or the cohesiveness of individual clones, we developed tools. Our clone-refactoring tools were evaluated through an extensive user study, where programmers could take actions based on automated analyses of the evolution of clone groups.

This process enhances software quality through refactoring, and we utilized the data to create a procedure for selecting clones that considers various factors. Introducing CREC, a machine-learning technique that recommends software replication by mining features from new and historical data, considering the existing code base. CREC is a cross-project technique that can recommend existing clone groups, supporting a comprehensive approach to refactoring.

To assemble the training set, CREC automatically extracts historically refactored clone groups (R-clones) and those that have not been refactored from a specified set of software repositories. CREC extracts 34 attributes characterizing the content and evolution behaviors of individual clones, along with spatial, syntactical, and co-change relations of clone peers. This information is used by CREC to train a classifier for recommending which copies should be refactored next. Our cutting-edge clone suggestion toolset achieved an 83 percentage F-score for a single project and 76 percentage for multiple projects. Compared to the state-of-the-art similar approach, CREC showed higher effectiveness, providing valuable insights for future developments.

”The use of refactoring strategies for clone maintenance in the context of method- level code clones”: The researchers primarily focused on issues like clone maintenance to assist coders. Refactoring reduces time spent on software maintenance, enhancing readability, structure, performance, abstraction, and maintainability. This research contributes to a streamlined strategy for clone maintenance by focusing on clone modification. A copy-and-paste refactoring strategy was tweaked using the program ”Clone Manager” to track and plan for future modifications.

The enhanced tool, tested on open-source projects, was compared with three other applications for evaluation purposes. Through these efforts, a more efficient strategy for the processes involved in clone maintenance was developed, addressing persistent issues in software development.

3 Methodology

The methodology for creating a categorization and outlier detection model for code clones

involves several key steps. Initially, the test cases are organized into groups with known and unknown sizes. To assess the classifier model, a meticulously specified dataset is employed, contributing significantly to the model's favorable results. The subsequent sections provide a comprehensive explanation of the proposed strategy.

3.1 Test Case Organization and Dataset Specification:

The initial phase involves grouping test cases into known and unknown sizes. A critical aspect of the methodology is the testing and validation of the classifier model, accomplished using a meticulously specified dataset. The subsequent sections provide a detailed explanation of the proposed strategy.

3.2 Outlier Detection Techniques:

A pivotal step in the methodology is the identification of outliers in the data. Following this, a model is trained for closed-set classification, and its performance is assessed on the open-set. The use of dataset vectors in both training and testing phases, generated by running multiple instances of the code under analysis, adds a layer of robustness. The Local Outlier Factor (LOF) approach is introduced to evaluate the likelihood of a data point being an outlier.

3.3 Code Tokenization and Analysis:

To address code analysis challenges, the methodology employs machine learning methods after dividing each piece of code into distinct tokens. Utilizing modern lexical and syntactic analysis tools, including abstract syntax trees, reveals only the structures found in the source code.

3.4 Feature Extraction and Vector Creation:

Crucial to the methodology is the analysis of code functionality using the Java Development Kit (JDT). Pairs of instances are created from feature vectors, and the local outlier factor approach is applied to detect anomalies in the feature vector dataset.

3.5 Machine Learning Models and Refactored Type Recognition:

The methodology explores machine learning models, defining the concept and its various applications. It involves the development of a system for detecting refactored types using machine learning, recognizing refactored types constructed through these techniques.

3.6 Training and Evaluation Phases:

Following a two-phase approach, the system is trained on a predetermined code corpus and then tested on a separate dataset. Evaluation is conducted with well-known classification models such as Bagging, K-nearest neighbors (KNN), Forest PA, and Random Forest models. The effectiveness of supervised learning classifiers is assessed through cross-validation.

3.7 Automated Refactoring Clone Prediction:

This phase includes the creation of the automated refactoring clone prediction model. The model is applied to software projects, and outcomes are experimentally assessed. The effectiveness of supervised learning classifiers is further examined using a 10-fold cross-validation approach.

3.8 Comparison and Decision-Making:

The methodology concludes with a comparison of findings with a prior study employing similar techniques. This leads to decisions regarding restructuring options for different clone types (Refactoring and Unknown).

The decision to employ a neural network model as the top performance and the visualization techniques support our goal of developing a user-friendly application that reliably predicts calories burned and advances the field of calorie tracking by utilizing a variety of methods.

4 Algorithms

```
begin
  for i := 1 to 10 s
      tep 1 do expt
      (2, i);
      newline() od
where
proc expt(x, n) ≡
  z := 1;
  do if n = 0 then exit fi;
     do if odd(n) then ex
           it fi;
    n := n/2; x := x ∗ x od;
      { n > 0 };
      n := n − 1; z
  := z ∗ x od; p r i n t
  (z).
end
```

Algorithm 1
Calculate $y = x$

Require: $n \geq 0 \vee x \neq 0$ **Ensure:** $y = x$

1: $y \Leftarrow 1$
2: **if** $n < 0$ **then**
3: $X \Leftarrow 1/x$
4: $N \Leftarrow -n$
5: **else**
6: $X \Leftarrow x$
7: $N \Leftarrow n$

```
8: end if
9: while N /= 0 do
10:     if N is even then
11:         X <= X × X
12:         N <= N/2
13:     else[N is odd]
14:         y <= y × X
15:         N <= N − 1
16:     end if
17: end while
```

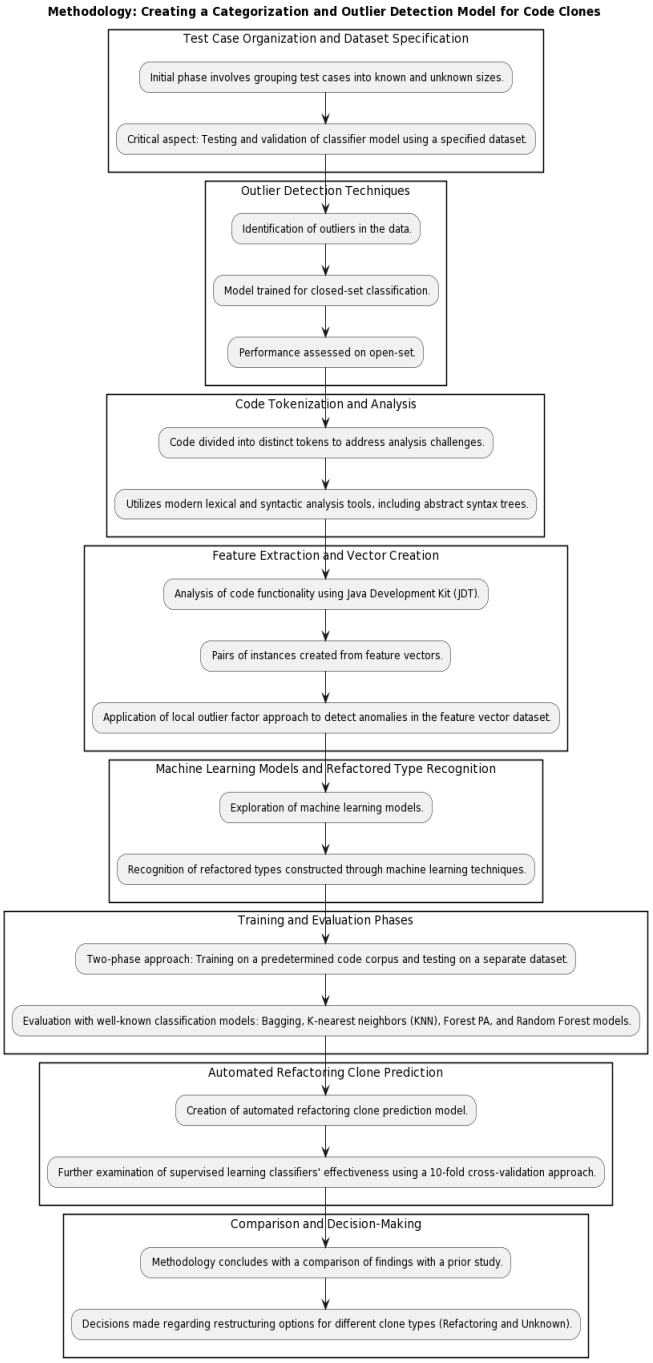

Fig. 1 Flowchart illustrating the Proposed Methodology for Automated Code Clone Detection and Refactoring, showcasing the step-by-step process of feature extraction, hybrid machine learning algorithms, outlier reclassification, and comprehensive performance evaluation.”

5 Results

Based on our study, utilizing machine learning as an automatic refactoring advisor proves effective in reducing defects in cloned software. Our results demonstrate its capability to automatically identify clones in legacy systems, including the detection of clones that were previously overlooked.

After transferring your dataset, the next step involves generating feature vectors. To calculate the local outlier factor, select the rows and columns for elimination and click the corresponding button. Each feature vector column is assigned a value between
-1 and 1, with the rows having the highest value marked as outliers. The remaining rows and columns are considered the norm, where 1 indicates the column is unnecessary and -1 indicates it contains irrelevant information. To assess the model's predictions, utilize the "Run KNN Algorithm" and other algorithm buttons to build and train the model on the dataset. Local outlier factors help identify redundant columns containing essential data. The "Comparison Graph" button provides a visual representation of accuracy, precision, recall, and other measures. Running the algorithm again on the modified dataset yields updated results.

The comparison graph illustrates the performance of each algorithm, with the random forest consistently leading in every category. Once machine learning models are completed, users can click the "Refactor Software Advisor" button to access a list of applications requiring refactoring. Upon completion of the analysis, users receive a detailed report with suggestions for applications that may need refactoring. If an application has already undergone refactoring, no additional refactoring is necessary.

In a previous section, we identified classes requiring refactoring; now, we can open the file AnnotationBinding.java to search for instances of duplicate code.

Remarkably, the same software employs identical code for two separate procedures, each involving a distinct set of keys in the obtaining Keys process.

6 Conclusion

In this study, we propose a learning approach to automatically extract features from identified code clones and reduce the number of duplicated lines of code by introducing refactors into software systems. The suggested method utilizes a hybrid of machine- learning algorithms to identify code clones and automatically recommend suitable refactoring in the implemented project. This aims to predict code clone detection and facilitate the removal of clones. Based on experimental results, the proposed approach demonstrates promising outcomes, potentially aiding developers in mitigating code clone-related issues. Its implementation in a real project is anticipated to yield even better results. We also present a comprehensive framework proficient in identifying and removing software code clones using a machine-learning algorithm.

To enhance classification precision, we introduce a novel approach to reclassify clone-type outliers as Unknown clones. The primary advantage is its ability to handle any type of exception without requiring additional effort from developers. This incorporation of inferred outlier information enhances classification accuracy and pro- vides valuable insights for detecting various types of code clones. The reliability of our proposed approach is affirmed by the quality of the results.

We conduct a thorough comparative study to assess the performance of our proposed method against leading-edge classification tools. We evaluate accuracy, reliability, and robustness under varying datasets and different configurations.

Utilizing our distinct tools, we underscore the importance of reducing and eliminating

code clones in our developed software system. Beyond minimizing code duplication and improving overall system efficiency, eliminating code clones can contribute to reducing the costs associated with system maintenance.

We introduce a novel machine learning framework that automatically extracts features from detected code clones and trains models to differentiate between refactored and non-refactored clone code segments. The objective is to develop an auto- mated analysis tool that provides developers with clone-specific information, enabling better-informed decisions regarding refactoring efforts. Our results demonstrate the effectiveness of this approach in differentiating likely code clones from those less likely, relying on semantic and morphological features. By investigating all aspects of code structure relevant to identifying potential duplications, the proposed model proves highly effective in differentiating between different types of cloned source code.

Experimental results showcase the superiority of our approach in detecting various code clones under different configurations and approaches. The approach's generalizability to detect various instances in different programming languages with minimal need for modifications underscores its potential for automatic detection in large code bases. Our method significantly improves clone accuracy by removing a high percentage of false negatives, addressing concerns about code quality and the inadvertent introduction of bugs.

In the future, we aim to expand the project's scope, potentially incorporating set classification and deep learning techniques. The development of a multi-threaded classifier capable of simultaneously processing multiple cloned code instances in parallel is also considered. By advancing the techniques demonstrated in this project, we hope to find solutions that minimize the negative impacts of poor-quality software on product development programs. Additionally, we plan to investigate other factors affecting developer productivity, such as the impact of tools and human factors on software code quality. Ultimately, our goal is to develop methods that enhance overall software quality while reducing development costs.

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. G¨ode and R. Koschke, "Frequency and risks of changes to clones," in Proc. 33rd Int. Conf. Softw. Eng., 2011, pp. 311–320.
6. 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.
7. 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.Sciences 12.14 (2022): 6967.
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. Challagundla, Yagnesh, et al. "Screening of Citrus Diseases Using Deep Learning Embedders and Machine Learning Techniques." 2023 3rd International conference on Artificial Intelligence and Signal Processing (AISP). IEEE, 2023.
11. Karthik, S., and B. Rajdeepa. quot;A collaborative method for code clone detection using a deep learning model.quot; Advances in Engineering Software 174 (2022): 103327.
12. Nadim, Md, et al. quot;Evaluating the performance of clone detection tools in detecting cloned co-change candidates.quot; Journal of Systems and Software 187 (2022): 111229.
13. Lei, Maggie, et al. quot;Deep learning application on code clone detection: A review of current knowledge.quot; Journal of Systems and Software 184 (2022): 111141.
14. Kanwal, Jaweria, et al. quot;Historical perspective of code clone refactorings in evolving software.quot; Plos one 17.12 (2022): e0277216.
15. Lei, Maggie, et al. quot;Deep learning application on code clone detection: A review of current knowledge.quot; Journal of Systems and Software 184 (2022): 111141.
16. Allamanis, Miltiadis. quot;The adverse effects of code duplication in machine learn- ing models of code.quot; Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. 2019.