Intelligent Computing Technology and Automation
Z. Hou (Ed.)

doi:10.3233/ATDE231264

A Software Code Infringement Detection Scheme Based on Integration Learning

Meng QIN[a,1]
[a] *Department of Law and Economic Management, Hulunbuir University, Hulunbuir, 021008, China*

Abstract. A software code plagiarism detection scheme based on ensemble learning is designed to address the issue of low accuracy in traditional abstract syntax tree based software code infringement detection methods. We adopt the AST structure of the code to integrate domain partitioning in IR with AST, and use a weighted simplified abstract syntax tree to design feature extraction and similarity calculation methods, to achieve partial detection of semantic plagiarism and calculate the similarity between text and source code. Then, the feature set of the known classification training set is placed into a random forest based ensemble classifier for training, and an association between error rate and the classification effect of the decision tree in the random forest are proposed to acquire feature node matching with the feature in the code base. The experimental results show that our scheme has higher accuracy than traditional detection methods based on abstract syntax trees. It can not only detect code similarity, but also provide the types of plagiarism, which has better comprehensive identification performance.

Keywords. code infringement; AST; plagiarism detection; integration learning; SVM

1. Introduction

The source code, as the core of software, is the result of the hard work of developers, as well as the crystallization of wisdom and art. However, the universality of some software in terms of functionality leads to plagiarism not only in the design level, but also in the underlying code, resulting in frequent software infringement cases. Software developers often refer to the code of others when coding software, especially obtaining a large amount of code through the internet. A large amount of open source code has brought great convenience to this development model, but due to the diverse types of licenses used for open source code, the constraints of each license are different. Blindly reusing software is likely to bring serious legal risks to enterprises. Due to the large number of enterprise software developers and code volume, as well as the countless number of open-source software, manual inspection of various licenses and infringement risks is almost impossible. Therefore, in order to help enterprises avoid potential legal risks, it is necessary to have a method and system that can detect and report potential software infringement risks. The homology analysis technology of source code can be used to compare the similarity between codes. This technology has been widely used in various fields such as code plagiarism, software copyright protection, vulnerability mining, etc. Currently, typical representatives of code similarity detection software at home and abroad include Sim, Moss, Yap, Jplag,

[1] Corresponding Author: Meng QIN; Department of Law and Economic Management, Hulunbuir University, Hulunbuir, 021008, China; yyuereentg@126.com

BuaaSim, and CCFinder. [1]. Literature [2] summarizes different approaches to automatically detecting similar code and categorizes them into four methods: text based, syntax tree based, graph based, and metric based, all of which are mostly pipeline based [3-5]. In recent years, ensemble learning has also become a commonly used method for model aggregation. Multiple model ensemble methods can further improve the expression ability of models and unleash the potential of features. However, as is well known, the detection ability of models is not only limited by the model's expressive ability, but also by the potential ability of features. Some experts construct multiple model ensemble based on multiple features to elevate the upper limit of feature potential, Make the model's expressive power no longer limited to single features.

This article proposes a software code infringement detection method based on AST and ensemble learning classification, aiming to improve the efficiency and accuracy of code plagiarism detection. We first clarified the concept of text similarity and focused on discussing the application process of AST in information retrieval methods. On this basis, a preliminary study was conducted on the data preprocessing process. Comprehensively consider the characteristics of vocabulary, structure, and code style to describe and compare the characteristics of different codes more comprehensively, in order to analyze and evaluate code pairs. Then, the similarity alignment method is used to construct a feature sequence, and suitable similarity calculation formulas are selected based on specific requirements and the representation of the code feature set. Finally, the improved random forest was applied for random sampling and training to obtain a classification with similar codes. The experimental part uses the resource library of student program assignments as a case study to extract feature sets from the code using the previously mentioned method. The results indicate that the method proposed in this article has improved time efficiency and accuracy in detecting source code plagiarism, and can be effectively used for software code infringement identification.

2. Program Similarity Calculation Method

From the analysis of the technical characteristics of various existing code plagiarism detection systems, it can be seen that a single method cannot effectively detect plagiarism. In order to reduce the error rate of existing detection methods, this article comprehensively uses text analysis, structural measurement, and attribute counting techniques to calculate the text similarity, structural similarity, and variable similarity of the program. Based on the weight of these three similarities, the final overall program similarity is obtained to determine whether there is plagiarism. Through experiments, it has been confirmed that this method can effectively detect various plagiarism behaviors, and the accuracy and recall of the detection results have been improved. Especially for detecting plagiarism behavior in program assignments with simple structure in programming courses, it has stronger practicality [6].

In the process of software code infringement detection, we borrow the method from literature [7] and use a combination of AST in IR and random forest to conduct it, as depicted in figure 1. The former can quickly retrieve potential plagiarism code pairs from massive source libraries, but it is basically unsupervised, so when using datasets with classification labels, these prior knowledge cannot be applied. In addition, the potential plagiarism code pairs after retrieval only indicate that there is suspicion of

plagiarism in these code pairs. To truly determine whether plagiarism is true, a more granular analysis of the characteristics of the code pairs is needed.

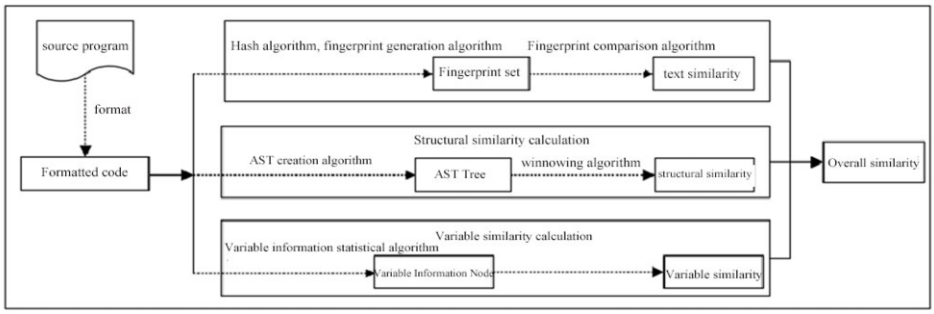


Figure 1. Program similarity algorithm calculation process.

3. Software Industry Code Infringement Detection Based on Integration Learning

3.1 Data Preprocessing

To eliminate confusion in code plagiarism, AST preprocessing can be performed to eliminate the effects of changing comments, spaces, and variable names [8]. The preprocessing content includes: (1) traversing the AST and removing annotation nodes from the AST. Annotations usually do not affect the structure and logic of the code, so they can be ignored. (2) Replace all variable names with unified placeholders. This can eliminate differences in variable names, making the structure and logic of the code easier to compare. (3) Replace all identifiers (such as function names, class names, etc.) with unified placeholders. This can eliminate differences in identifiers and make the structure and logic of the code easier to compare. (4) Ignore surface differences such as changes in comments, spaces, and variable names, and remove comment nodes from AST. Annotations usually do not affect the structure and logic of the code, so they can be ignored. The specific replacement rules are shown in Table 1:

Table 1. Source code character replacement rules

Replacement Rules	Replacement object	Replace with
Annotation Replacement Rule	Annotation Node	"COMMENT"
Space Replacement Rule	Space Node	"SPACE"
Variable Name Replacement Rule	Variable Name Node	"VAR"
Constant Replacement Rule	Constant Value Node	"CONST"
Identifier replacement rule	identifier node	"IDENTIFIER

3.2 Construction Feature Sequence

Using the JDT (Java Development Tools) tool suite of the Eclipse platform to automatically extract abstract syntax trees (ASTs) from Java programs in the code set. In the construction path of the project, introduce the JDT library to use the AST parsing function provided by JDT. We can add JDT libraries by right-clicking on the project, selecting "Build Path" ->"Configure Build Path", and then on the "Libraries" tab. We first created an ASTParser object and set the source code of the code file to be parsed. Then, parse the AST by calling the createAST method and use the ASTVisitor to

process the various nodes of the AST [9, 10]. To reduce computational overhead, it is possible to consider deep traversing the AST and converting the node sequence into a string feature sequence. This can encode the structure and information of AST into a string, facilitating similarity comparison and matching. Part of the key codes for this process are described as follows:

```
import ... org.eclipse.jdt.core.dom.*;
public class ASTFeatureExtractor {
private StringBuilder featureSequence;
public String extractFeatures(String code) {
    // Create ASTParser
    ASTParser parser = ASTParser.newParser(AST.JLS14);
    parser.setKind(ASTParser.K_COMPILATION_UNIT);
    //set source code of ASTParser
    parser.setSource(code.toCharArray());
    // resolv AST
    CompilationUnit cu = (CompilationUnit) parser.createAST(null);
    // Initialize feature sequence
    featureSequence = new StringBuilder();
    // depth-first traversal
    cu.accept(new ASTVisitor() {
        @Override
        public boolean visit(SimpleName node) {
            // Add node information to feature sequence
            featureSequence.append(node.getIdentifier()).append(" ");
            return super.visit(node);
        }
        // Rewrite the visit method for other types of nodes as needed to add
node information to the feature sequence
    });
    // Return feature sequence
    return featureSequence.toString();
}
}
```

By converting AST into a string feature sequence, the computational cost of subtree matching can be reduced while preserving the structure and information of the code. This allows for more efficient similarity comparison and matching, thereby improving the efficiency of code similarity detection.

3.3 Code Similarity Matching Strategy

Random forest is an ensemble learning algorithm based on decision trees, which improves the accuracy of classification or regression by integrating the prediction results of multiple decision trees Build a set of decision tree models using the random forest algorithm. By using out of pocket errors to assign weights to random forest decision trees, the model's attention to difficult to classify samples can be increased, thereby improving the model's generalization ability and accuracy. The approach in this article is to predict each decision tree using samples not selected by the decision tree and calculate the prediction error. The out of pocket error is the average error of all decision trees. Calculate the weight based on the OOB error of each decision tree [11].

The reciprocal of out of pocket errors can be used as weights, meaning that the smaller the error, the greater the weight of the decision tree Those samples that have not been selected are called out of bag samples. Assuming there are m out of bag samples, use the corresponding decision tree to classify these m samples and calculate the OOB error.

The computation equation is:

$$obb\_err(i) = \frac{err\_num(i)}{obb\_num(i)} \tag{1}$$

where $err\_num(i)$ is the number of sample classification errors of the i_{th} classifier; $obb\_num(i)$ is the samples number of out_of_bag, whose generalization error rate is the error rate on the training set, that is, an unbiased estimation of base classifiers. $obb\_err(i)$ is adopted to replace the posterior probability to simplify the computation as

$$obb\_err(i) = 1 - con(i)$$

Then the weight equations can be transformed as:

$$weight(i) = 1 - \frac{1/(1-obb\_err(i))}{\sum_{j=1}^{T} 1/(1-obb\_err(j))} - \frac{T-2}{T} \tag{2}$$

Compute the following $obb\_err(i)$ of n decision trees and assign them corresponding weight as $WRF[1]-WRF[n]$. The samples are classified by $WRF[1]-WRF[n]$ for the testing set and the classification results are recorded by each decision tree. For each classification result it is performed weighted statistics to acquired the final prediction result

$$pred = \{pred_1,\ pred_2,...,pred_m\} \tag{3}$$

4. Simulations

The dataset used in the experiment was selected from the public software libraries NPM (Node Package Manager) and PyPI (Python Package Index), involving a total of 6100 C# program code files. Randomly select a certain number of program code files from the entire dataset as samples. Ensure that the number of samples is large enough to represent the characteristics of the entire dataset. Manual judgment: Each sample program code file is manually judged by professional personnel or domain experts. They will check each sample file for plagiarism and attempt to determine the means of plagiarism Record the plagiarism judgment results and classification of each sample program code file for subsequent analysis and evaluation. The experimental data includes two parts: the code set of the program to be tested contains 2000 samples of program code that require plagiarism detection. These samples can come from different sources, such as student assignments, open-source software libraries, or commercial software, to evaluate the performance and accuracy of plagiarism detection algorithms in practical applications; The known plagiarism sample set contains a total of 500 program code samples that have been determined to have plagiarism behavior. These samples can be determined by experts or manual judgment, or come from known plagiarism cases.

Based on the experimental dataset, set the similarity ratio coefficient and threshold to adjust the accuracy of the algorithm. The similarity ratio coefficient is used to measure the similarity between two program codes, while the threshold is used to determine when two program codes are judged as plagiarism. As shown in Table 2, in order to evaluate the impact of different parameter settings on algorithm performance and select the optimal parameter combination to improve the accuracy of plagiarism detection, this article sets a similarity ratio coefficient α1, α2. The similarity threshold between two program codes is θ, which indicates when two program codes will be judged as plagiarism. Comparing the AST-SVM proposed in this article with the AST plagiarism detection method, the detection accuracy of different types of plagiarism codes is shown in Table 3. From it, it can be seen that the accuracy of AST-SVM in detecting code plagiarism behavior for Type A and Type B types is higher than 0.8, while the accuracy of detecting Type C and Type D types is relatively low, with 0.772 and 0.643, respectively. It can be seen that our scheme is effective in detecting different types of code infringement.

Table 2. Parameter setting for similarity comparison experiment

Similarity ratio	θ=0.25	θ=0.5	θ=1
α1=0， α2,=1	0.392	0.478	0.472
α1=0， α2,=0.25	0.456	0.518	0.335
α1=0.25， α2,=5	0.661	0.721	0.483
α1=0.5， α2,=75	0.702	0.867	0.607
α1=1， α2=0	0.531	0.664	0.528

Table 3. Detection results of different types of plagiarism rates

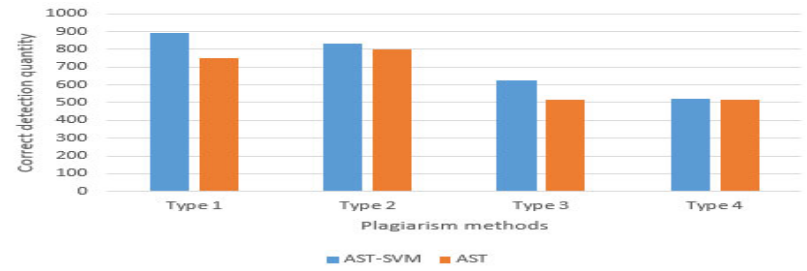
Algorithm	Type A	Type B	Type C	Type D
AST	0.814	0.905	0.765	0.587
AST-SVM	0.776	0.873	0.772	0.643

In order to further test the comprehensive performance of the algorithm, two program code files are randomly selected from the dataset for combination. Repeat this step until a sufficient number of composite samples are obtained. Extract a portion from each combination sample as the text set of the program code to be tested, and use the N-gram method to extract features.

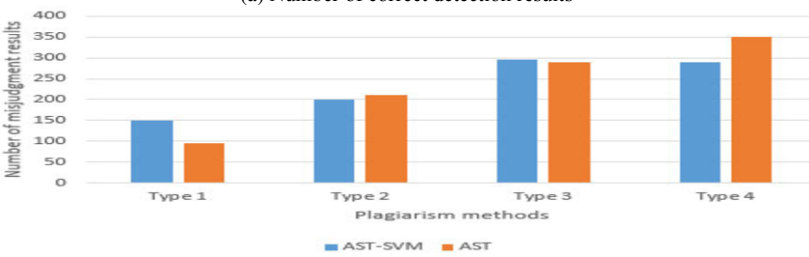
- Firstly, perform rough segmentation on the document to obtain segment sequences.
- Perform gram segmentation on segment sequences to obtain a list of gram frequencies. And select gram segments with frequencies greater than the set threshold as feature vectors.
- Each gram fragment is a dimension, forming a feature vector table.

For each detected program code text, calculate the Manhattan distance between them, and set a threshold based on the Manhattan distance calculation to determine whether two program code texts are considered plagiarism. Figures 2 shows the

detection error rate and recall rate of two different algorithms under the same conditions. It can be concluded that the detection accuracy of AST-SVM for Type l, 2, and 3 plagiarism methods is higher than that of AST model, and the overall recall rate has also increased by nearly 3.25%. Compared to traditional text feature based methods, AST-SVM can better capture the structural and semantic information of program code. Due to the fact that abstract syntax trees can reflect the hierarchical structure and syntax rules of code, they can more accurately determine the similarity and plagiarism relationship between two pieces of code.



(a) Number of correct detection results



(b) Number of misjudgment results

Figure 2. Comparison of the comprehensive performance of two algorithms for different types of plagiarism methods.

In addition, although this method uses N-grams to represent program code, which can improve the detection accuracy of plagiarism methods for reordering and adding operation types, if the plagiarist performs more reordering and adding operations on the code, it will affect the generated N-gram set and frequency. The N-gram set and frequency in the original code may not fully cover the newly added code portion, resulting in a decrease in detection accuracy. For n-grams, we may need to know that the larger the corpus size, the more useful the n-grams are for statistical language models, or the n-size of n-grams also has a significant impact on performance. Therefore, in the future, more complex plagiarism detection algorithms can be considered, such as deep learning based methods for learning more complex code patterns and structures, thereby improving the detection ability for reordering and new operations.

5. Conclusion

This article attempts to use machine learning techniques to train a model to identify infringement behavior in source code. This method can train a large amount of source code to enable the model to learn different types of infringement patterns and detect

them in new source code. We have provided a detailed explanation of the overall structure of existing code plagiarism detection, and conducted in-depth research on the classification of features based on IR retrieval and AST. On this basis, it is proposed to use the AST+SVM method to structurally represent program code through an abstract syntax tree, in order to better capture the hierarchical structure and syntax rules of the code. The experimental results show that compared to text feature based methods, this structured representation can more accurately depict the semantic information of the code, which improves the overall recall rate of plagiarism detection, and it also has advantages especially when dealing with structured program code.

References

[1] Cai Liang，Zhang Gang，Zhao Fengyu. Method and Implementation of Code Infringement Detection Based on Local Proxy and Index Information. Software Guide, 2017, 16(6): 5-10.

[2] Xin-Juna W U, Yub L. Research on Code Plagiarism Detection and Application in the Experimental Teaching. Experiment Science and Technology, 2011, 9(5): 275-278

[3] Cao Yuzhong, Jin Maozhong, Liu Chao. Overview of Clone Code Detection Technology. Computer Engineering and Science，2006，28 (A2) : 9-103.

[4] Han M L, Kwak B, Kim H K, et al. Implementation of the Personal Information Infringement Detection Module in the HTML5 Web Service Environment. Journal of the Korea Institute of Information Security & Cryptology, 2016, 26(4):1025-1036.

[5] Matefi R, Musan M. ECJ case law and its impact on the evolution of administrative liability; state liability for infringement. Bulletin of the Transilvania University of Braov, 2011, 4(1):4.

[6] YANG Chao. Hybrid plagiarism detection method in program code based on multiple techniques. Computer Engineering and Applications, 2016, 52(18): 222-227.

[7] Zhang Min. Source Code Plagiarism Detection Based on Information Retrieval and Stacking Integrated Learning [D]. Beijing Jiaotong University, 2023.

[8] Jinyuan Z, Yuanguang T, Shudong Z. Integrated design of target detection system based on marine data buoy. Meteorological,Hydrological and Marine Instruments, 2013, 30(2):73-76.

[9] Jin X, Luo J, Yu J, et al. Reinforced Similarity Integration in Image-Rich Information Networks. IEEE Transactions on Knowledge and Data Engineering, 2013, 25(2):448-460.

[10] A, Alena G. Esposito, and P. J. B. B. Self-derivation through memory integration under low surface similarity conditions: The case of multiple languages. Journal of Experimental Child Psychology, 2019, 187:104661-104661.

[11] Gordon P C, Hendrick R, Johnson M, et al. Similarity-based interference during language comprehension: Evidence from eye tracking during reading. Journal of Experimental Psychology Learning Memory & Cognition, 2006, 32(6):1304.