Identifying Auto-Generated Code by Using Machine Learning Techniques

Kento Shimonaka*, Soichi Sumi*, Yoshiki Higo* and Shinji Kusumoto*
*Graduate School of Information Science and Technology
Osaka University, Japan
Email: {s-kento, s-sumi, higo, kusumoto} @ist.osaka-u.ac.jp

***Abstract*—Recently, many researchers have conducted mining source code repositories to retrieve useful information about software development. Source code repositories often include auto-generated code, and auto-generated code is usually removed in a preprocessing phase because the presence of auto-generated code is harmful to source code analysis. A usual way to remove auto-generated code is searching particular comments which exist among auto-generated code. However, we cannot identify auto-generated code automatically with such a way if comments have disappeared. In addition, it takes too much time to identify auto-generated code manually. Therefore, we propose a technique to identify auto-generated code automatically by using machine learning techniques. In our proposed technique, we can identify whether source code is auto-generated code or not by utilizing syntactic information of source code. In order to evaluate the proposed technique, we conducted experiments on source code generated by four kinds of code generators. As a result, we confirmed that the proposed technique was able to identify auto-generated code with high accuracy.**

I. Introduction

Source code analysis is one of the main research fields of software engineering. For example, a significant amount of research has been conducted on finding similar parts from source files and mining code changes in source code repositories. Source code of software systems often includes auto-generated code [1] [2], i.e., output of parser generators. The presence of auto-generated code occasionally hinders source code analysis. For example, if we detect code clones from source code including auto-generated code, a large number of code clones are detected from auto-generated code [3] [4]. Code clones detected from handmade code should not be paid attention because their amount is much smaller than the code clones detected from auto-generated code. That may cause developers miss some important findings by clone detection tools. Another example is mining source code repositories. In mining source code repositories, code elements such as classes and methods are tracked through thousands of revisions. The presence of auto-generated code requires longer time to track code elements [5].

In auto-generated code, there are usually code comments that represent they are auto-generated code. Thus, we can find auto-generated code by using search tools such as UNIX *grep* and remove them. However, such code comments are occasionally deleted by programmers. In order to find and remove auto-generated code without code comments, all we can do is checking source files manually one by one. Consequently, in this paper, we propose a technique to identify auto-generated code automatically even if they do not have code comments.

In order to identify auto-generated code without code comments, we try to grasp their features that are not common to manually developed code. However, it is unrealistic to find such features by hand work, and so we determined to use machine learning techniques to collect such features automatically. First we collected auto-generated source files manually, then we made some machine learning models by using the auto-generated source files. Besides, we have conducted some experiments with the models. The experimental results showed that we were able to identify auto-generated code automatically with very high accuracy.

The remainder of this paper is organized as follows: Section II explains the proposed technique and Section III describes the experiments. Sections IV and V describe discussion of the experiments and some threats to validity, respectively. Finally, we conclude this paper in Section VI.

II. Proposed Technique

In this paper, we propose a technique to identify auto-generated code automatically by using machine learning techniques. Figure 1 illustrates an overview of the proposed technique. The proposed technique takes learning data (which consists of auto-generated source files and handmade source files) and test data as its input, and predicts whether the test data is auto-generated code or not as its output.

Machine learning techniques learn features of learning data and construct a learning model to predict unknown classes of test data. By providing the learning model with syntactic information of test data, the test data is predicted whether it is auto-generated code or not. Each feature of learning data is called explanatory variable, and each class of test data (which is to be predicted) is called objective variable.

The proposed technique consists of following two steps. In Step 1, we obtain syntactic information of learning data, then construct a learning model by using the syntactic information. In Step 2, we apply the learning model to test data which is to be predicted whether it is auto-generated code or not. In the remainder of this section, we describe each step in detail.

A. Step 1: constructing a learning model

In Step 1, we obtain syntactic information of learning data. As the syntactic information, we use AST (Abstract

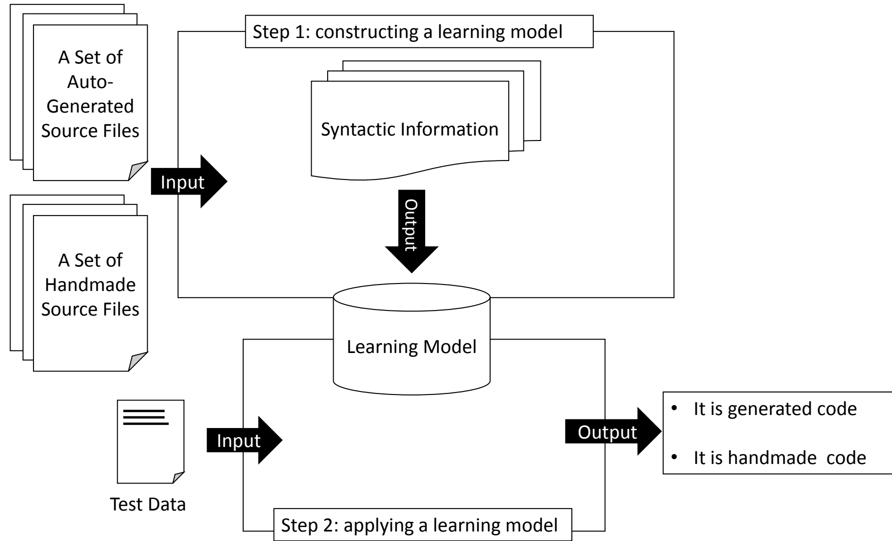

Fig. 1. Overview of The Proposed Technique

Syntax Tree) nodes of source code. The value that explanatory variables take is the number of occurrences of each AST node, or integer value. The objective variable indicates whether given source code is auto-generated code or not. In Section III, we give an example of AST of Eclipse JDT, which is a library we used in our implementation.

After obtaining the syntactic information, we construct a learning model based on the syntactic information. In constructing a learning model, prediction accuracy decreases if the number of explanatory variables is too large, which is called over-fitting. Thus, we do not use all of the explanatory variables, but we conduct a variable selection to leverage only useful ones. We use four algorithms, Decision Tree [6], Random Forest [7], Naive Bayes [8], and SVM [9], to construct learning models. These algorithms are representative ones on machine learning techniques. Table I illustrates features of each algorithm.

TABLE I
ALGORITHM FOR CONSTRUCTING A LEARNING MODEL

Algorithm	Description
Decision Tree	It generates a conditional branch tree following the values of the explanatory variables. It is useful for learning data which contains many missing values and outliers.
Random Forest	It generates some decision trees by using learning data sampled randomly. It combines them and generates a more precise decision tree. It takes more time than Decision Tree.
Naive Bayes	It uses conditional probability. It is one of the most basic algorithms on machine learning techniques, which takes less time.
SVM	It generates a classifier for 2 classes. It is one of the most precise classification algorithms [10].

B. Step 2: applying a learning model

In Step 2, we apply the learning model to unknown source files. For this purpose, we prepare source files as test data which is to be predicted whether it is auto-generated code or not. We obtain syntactic information of the test data like Step 1. Furthermore, we conduct a variable selection. Selected variables are the same as the ones that were selected in Step 1. After the above process, the learning model predicts whether the test data is auto-generated code or not.

III. Experiment

In this section, we describe four experiments that we conducted to evaluate the proposed technique. The followings are short descriptions for the four experiments.

Experiment 1: We evaluated the learning models as Step 1 of the proposed technique. To evaluate them, cross-validation was conducted.

Experiment 2: We evaluated the learning models in the same way as experiment 1, but we used auto-generated source files and handmade source files that are larger than 10 KBytes.

Experiment 3: We applied learning models made with source files of a code generator to ones of other code generators.

Experiment 4: We applied a learning model to unknown source files, and confirmed if auto-generated code was identified.

A. Experimental Object

We used source files generated by four kinds of code generators in the experiments. Table II illustrates names of the four kinds of code generators and the number of files of auto-generated source files. These code generators are parser generators for Java. Collected source files are written in Java. As for test data, we used *UCI Source Code Data Sets* [11] (hereafter, we call it *UCI datasets*).

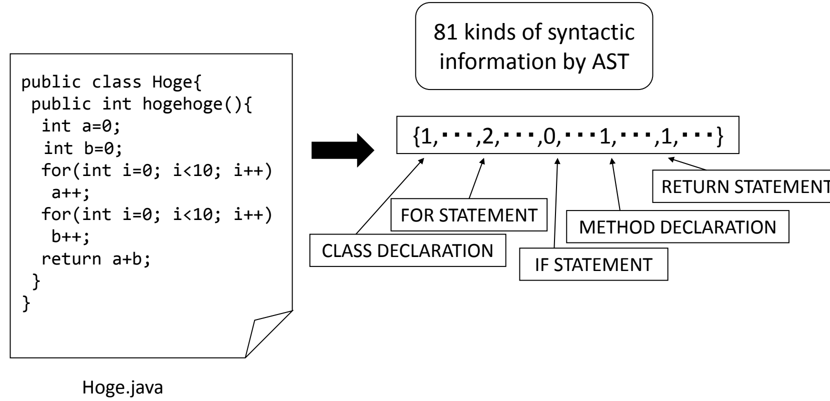

Fig. 2. Example of AST Node Generation

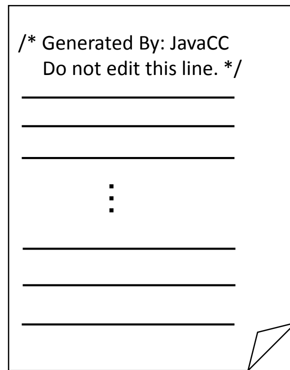

Fig. 3. Example of Code Comments in Auto-Generated Code

To obtain syntactic information, we used Eclipse JDT 3.10 [12] to generate ASTs. Figure 2 illustrates an example of generating AST nodes. Eclipse JDT 3.10 defines 84 kinds of AST nodes, and we used them as explanatory variables, other than three nodes related to code comments. In other words, the number of explanatory variables is 81.

In conducting a variable selection, constructing the learning models and predicting the test data, we used Weka [13], which is the library developed by Java for machine learning techniques.

TABLE II
AUTO-GENERATED SOURCE FILES

Code Generator	ANTLR	JavaCC	JFlex	SableCC
Number of Files	8,778	21,219	3,789	16,066

B. Data Collection

In order to construct learning models predicting whether given source code is auto-generated code or not, we collected auto-generated source files and handmade source files. This section describes the technique to collect the source files.

First, we collected auto-generated source files. Some auto-generated source files have code comments that indicate they are auto-generated code. Figure 3 is an example of such comments. The auto-generated source files were collected by text search. More concretely, auto-generated source files were automatically collected from GitHub [14] by web scraping using JSoup [15]. In this research, auto-generated source files changed by developers are regarded as auto-generated source files.

Second, we collected handmade source files. The handmade source files were collected from Apache repository [16]. The auto-generated source files in Apache repository were removed by text search. The handmade source files were collected as many as the number of auto-generated source files.

C. Evaluation Measures

We used precision and recall as evaluation measures of learning models. The following describes the definitions of the measures.

Precision: the ratio of source files that can be correctly predicted by a learning model to all source files.

Recall: the ratio of auto-generated source files that can be correctly predicted by a learning model to all auto-generated source files.

In experiments 1 and 2, we calculated precision and recall. In experiment 3, we only calculated recall since experiment 3 did not use handmade source files. In experiment 4, we only calculated precision.

D. Experiment 1

In experiment 1, five learning models were constructed. Four of them were constructed from four learning data. Each

learning data is a set of source files generated by one of the four code generators. The remaining learning model was constructed from all auto-generated source files which we collected. To evaluate those learning models, cross-validation was conducted. In cross-validation, the original data set is partitioned into N equal size subsets. $N-1$ subsets are used as training data and the remaining one subset is used as test data. This process is repeated N times, and the output is an average of precision and recall of each iteration. Each subset is selected once as the test data.

Table III shows the results of the experiment 1. P and R indicate precision and recall, respectively. All files indicate the results when the learning model was constructed from all of four learning data. In many cases, precision and recall are larger than 90%. Especially, the results of ANTLR and JFlex are larger than 99% in most cases. The results of Random Forest are higher than the others and the results of Naive Bayes are lower than the others in most cases.

E. Experiment 2

In experiment 2, we investigated whether the performance of a learning model changes when a file size of the learning model is limited. Experiment 2 is similar to experiment 1. The difference between experiment 1 and experiment 2 is whether only large files are used or not. More concretely, we used 10 KBytes or larger files, which are approximately 400 or more lines of code. Table IV shows the number of files whose size is 10 KBytes or larger. Table V shows the results of the experiment 2. Underlines indicate that the results are higher than the ones in experiment 1. In most cases, precision and recall are higher than 90%. The results of Random Forest are higher than the others and the results of Naive Bayes are lower than the others in most cases.

TABLE IV
AUTO-GENERATED SOURCE FILES THAT ARE LARGER THAN 10 KBYTES

Code Generator	ANTLR	JavaCC	JFlex	SableCC
Number of Files	8,686	6,661	3,786	860

F. Experiment 3

In experiment 3, we investigated whether a learning model constructed from one learning data can identify source files generated by other code generators. We applied the learning models to auto-generated source files. As with the experiment 2, the learning data used for constructing learning models were restricted to 10 KBytes or more. Table VI shows the results of the experiment 3. In most cases, the results of the experiment 3 are greatly lower than the results of the experiments 1 and 2. In contrast with the results of learning models generated by ANTLR, JavaCC and JFlex, are 60 to 90%, the results of a learning model generated by SableCC are 0 to 40%. We considered that features of the source files generated by SableCC totally differ with features of source files generated by the other three generators.

G. Experiment 4

In experiment 4, we investigated whether the proposed technique detects auto-generated source files for which the comments have been removed. Two of the authors manually validated the source files judged as auto-generated source files. We applied learning models to a part of *UCI datasets* because it would consume much time to validate the output of the learning models. The learning data used for constructing learning models were limited to 10 KBytes or more.

Table VII shows the results of the experiment 4. We used Random Forest to construct a learning model because its precision and recall was the highest in the experiment 2. The results show that the proposed technique detects auto-generated source files with 70% accuracy. We also investigated source files judged as auto-generated source files by the proposed technique. We extracted 1% of the source files judged as handmade source files by the proposed technique, and manually checked them. The results of the investigation

TABLE III
RESULTS OF EXPERIMENT 1

Algorithm	ANTLR		JavaCC		JFlex		SableCC		All Files	
	P	R	P	R	P	R	P	R	P	R
Decision Tree	98.6%	98.6%	93.9%	93.9%	99.7%	99.7%	97.3%	97.2%	96.3%	96.3%
Naive Bayes	97.9%	97.9%	83.4%	76.3%	99.1%	99.1%	85.3%	81.3%	78.5%	72.6%
Random Forest	99.0%	99.0%	94.5%	94.5%	99.8%	99.8%	97.5%	97.4%	97.1%	97.1%
SVM	98.3%	98.3%	84.1%	83.1%	99.5%	99.5%	87.2%	83.5%	81.4%	75.3%

TABLE V
RESULTS OF EXPERIMENT 2

Algorithm	ANTLR		JavaCC		JFlex		SableCC		All Files	
	P	R	P	R	P	R	P	R	P	R
Decision Tree	97.4%	97.4%	98.7%	98.7%	99.5%	99.5%	99.3%	99.3%	97.1%	97.2%
Naive Bayes	94.5%	93.8%	92.6%	92.1%	98.9%	98.9%	96.0%	95.3%	91.4%	79.5%
Random Forest	98.3%	98.3%	99.4%	99.4%	99.9%	99.9%	99.7%	99.7%	97.9%	97.9%
SVM	96.8%	96.8%	97.0%	97.0%	98.7%	98.7%	97.3%	97.3%	87.1%	88.9%

show that the proposed technique detects handmade source files with 98% accuracy.

IV. Discussion

First, we discuss the results of experiments 1 and 2. The accuracies of experiment 2 are higher than the ones of experiment 1, which implies that the larger the training source files are the higher the accuracies become. We also investigated false positives manually and found the followings.

- Many false positives were small files, most of them had only less than 10 explanatory variables.
- Many case entries and literals were included in most of the false positives.

The first finding is our motivation of conducting experiment 2. Most of the small source files were interface definitions and abstract class definitions. We can say that if a target source file does not include less than several program statements, it tends to be judged wrongly. However, such small files did not include many code elements, so that they were not likely to be obstacles of code clone detection and mining source code repositories. Many case entries and literals appeared in source files including functions of program analysis, which look like auto-generated code.

Second, we discuss the results of experiment 3. In most cases, models generated from a parser generator did not work well for auto-generated code of other parser generators. At this moment, we need to make models for each of parser generators to high accuracy identification. However, we are going to try to achieve high accuracy with cross parser generator models.

Last, we discuss the results of experiment 4. We investigated dozens of false positives, which were not auto-generated code regarded as auto-generated ones. The common feature of them were:

- they included many case entries and literals, and
- they did not include many other program elements.

In order to avoid such false positives, we need to consider adding other information than structural information.

TABLE VII
Results of Experiment 4

Number of Files	146,346
Generated source files which were predicted	438
Generated source files which were confirmed manually	304
Precision	69%

V. Threats to Validity

In the experiments, we collected many source files from the Apache repository. In case they did not contain code comments such as "generated by", they were treated as manually developed code. However, some of them might be auto-generated code in truth. This threat is categorized as a threat to construct validity.

We used four kinds of Java auto-generated code, which had been generated by four parser generators. Thus, if we use other programming languages or other parser generators, we might have different results. This threat is categorized as a threat to external validity.

In experiment 4, we manually checked if identified code was auto-generated one or not. We are not developers of the target software, so that we may have mistaken to judge them. However, we investigated with two persons to avoid such mistaking as much as possible. This threat is categorized as a threat to reliability.

VI. Conclusion

In this paper, we proposed a technique to identify auto-generated code automatically. The proposed technique utilizes structural information of program source code and some machine learning techniques to identify auto-generated code even if there are no code comments such as "generated by".

TABLE VI
Results of Experiment 3

Code generator	Algorithm	ANTLR	JavaCC	JFlex	SableCC
ANTLR	Decision Tree	-	62.6%	71.4%	38.3%
	Naive Bayes	-	63.4%	84.5%	32.6%
	Random Forest	-	64.6%	85.6%	45.3%
	SVM	-	49.7%	60.6%	23.8%
JavaCC	Decision Tree	75.2%	-	86.8%	22.0%
	Naive Bayes	85.7%	-	99.6%	31.1%
	Random Forest	75.3%	-	99.2%	24.3%
	SVM	71.8%	-	99.6%	34.6%
JFlex	Decision Tree	71.0%	75.4%	-	18.1%
	Naive Bayes	66.6%	92.1%	-	18.9%
	Random Forest	66.5%	51.6%	-	0.1%
	SVM	69.3%	88.0%	-	4.1%
SableCC	Decision Tree	1.2%	3.6%	0.0%	-
	Naive Bayes	22.0%	3.2%	0.1%	-
	Random Forest	0.4%	0.2%	0.0%	-
	SVM	2.7%	1.6%	0.9%	-

In the experiments, we used auto-generated code, which were generated by four parser generators. As a result, we confirmed that both recall and precision were over 90% in most cases. However, if we applied a model created by a parser generators to other parser generators, the accuracies dropped. In the future, we are going to conduct research to make robust models that can be applied to many kinds of auto-generated code with high accuracy.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Numbers 25220003, 24650011, and 24680002.

References

[1] McDonald, Pam, Dan Strickland, and Charles Wildman. "Estimating the effective size of autogenerated code in a large software project." Proceedings of the 17th International Forum on COCOMO and Software Cost Modeling. 2002.

[2] Uchida, Shinji, et al. "Software analysis by code clones in open source software." Journal of Computer Information Systems 45.3 (2005): 1-11.

[3] J. Harder and N. Goede, "Cloned code: stable code", Jour- nal of Software Evolution and Process 2013.

[4] Gode, Nils, and Rainer Koschke. "Frequency and risks of changes to clones." Proceedings of the 33rd International Conference on Software Engineering. ACM, 2011.

[5] MacLean, Alexander C., et al. "Trends that affect temporal analysis using sourceforge data." Proceedings of the 5th International Workshop on Public Data about Software Development (WoPDaSD ' 10). 2010.

[6] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. "Classification and regression trees.", CRS press 1984.

[7] Leo Breiman, "Random Forests.", Machine Learning 2001.

[8] Domingos, Pedro and Pazzani, "On the optimality of the simple Bayesian classifier under zero-one loss", Machine Learning 1997.

[9] V. Vapnik and A. Lerner, "Pattern recognition using generalized portrait method.", Automation and Remote Control, 24, 1963.

[10] Burges, Christopher JC. "A tutorial on support vector machines for pattern recognition." Data mining and knowledge discovery 2.2 (1998): 121-167.

[11] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi, "Uci source code data sets".

[12] Eclipse Java development tools (JDT). http://www.eclipse.org/jdt/.

[13] "Weka 3: Data Mining Software in Java", http://www.cs.waikato.ac.nz/ml/weka/.

[14] "GitHub", https://github.com/.

[15] "jsoup: Java HTML Parser", http://jsoup.org/.

[16] "Apache, Source code repository", http://svn.apache.org/repos/asf/.