On the Naturalness of Auto-generated Code —Can We Identify Auto-Generated Code Automatically?—

Masayuki Doi, Yoshiki Higo, Ryo Arima, Kento Shimonaka, and Shinji Kusumoto
Osaka University
Suita, Osaka, Japan
{m-doi,higo,r-arima,s-kento,kusumoto}@ist.osaka-u.ac.jp

ABSTRACT

Recently, a variety of studies have been conducted on source code analysis. If auto-generated code is included in the target source code, it is usually removed in a preprocessing phase because the presence of auto-generated code may have negative effects on source code analysis. A straightforward way to remove auto-generated code is searching special comments that are included in the files of auto-generated code. However, it becomes impossible to identify auto-generated code with the way if such special comments have disappeared for some reasons. It is obvious that it takes too much effort to see source files one by one manually. In this paper, we propose a new technique to identify auto-generated code by using the naturalness of auto-generated code. We used a golden set that includes thousands of hand-made source files and source files generated by four kinds of compiler-compilers. Through the evaluation with the dataset, we confirmed that our technique was able to identify auto-generated code with over 99% precision and recall for all the cases.

CCS CONCEPTS

• **Software and its engineering** → *Software creation and management*;

KEYWORDS

Auto-generated code, N-gram language model, Source code analysis

ACM Reference Format:
Masayuki Doi, Yoshiki Higo, Ryo Arima, Kento Shimonaka, and Shinji Kusumoto. 2018. On the Naturalness of Auto-generated Code —Can We Identify Auto-Generated Code Automatically?—. In *ICPC '18: ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3196321.3196356

1 INTRODUCTION

Recently, a variety of studies have been conducted on source code analysis. Targets of source code studies research are quite often open source software. The source code of open source software occasionally includes auto-generated code. If auto-generated code is included in the target software, it is usually removed in a preprocessing phase because the presence of auto-generated code may have negative effects on source code analysis. For example, if we conduct code clone detection on a set of source files including auto-generated code, a large number of code clones are detected from auto-generated code and code clones in hand-made code are indistinctive [7]. However in clone analysis situations, code clones in auto-generated code are rarely useful because they can be targets of neither clone merging refactoring nor simultaneous changes for fixing bugs. Another example is that presence of auto-generated code unnecessarily increases execution time for mining source code repositories [6].

A straightforward way to remove auto-generated code is searching special comments that are included in files of auto-generated code. However, it becomes impossible to identify auto-generated code with the way if such special comments have disappeared for some reasons. It is obvious that it takes too much effort to see source files one by one manually.

Shimonaka et al. proposed an automatic auto-generated code identification technique [9]. They created a learning model by using the information of co-occurring tokens in source code. Then, for a given unknown source file, four machine learning algorithms (decision tree, naive bayes, random forest, and support vector machine) were applied. In most cases, their technique worked well. Both precision and recall were over 90%. However, in some cases, such values became less than 90%.

Currently, we are conducting research on automatic identification of auto-generated code. In this paper, we propose a new technique to identify auto-generated code with the naturalness of auto-generated code. In our proposed technique, two stochastic language models are constructed: the first model is constructed with a set of hand-made code; the other model is constructed with known auto-generated code. Then, naturalness of unknown code is measured with the two models. If its naturalness as auto-generated code is higher than its naturalness as hand-made code, the unknown code is regarded as auto-generated code. We used a golden set that includes thousands of hand-made source files and source files generated by four kinds of compiler-compilers. Through the evaluation with the dataset, we confirmed that our technique was able to identify auto-generated code with over 99% precision and recall for all the cases.

2 AUTO-GENERATED CODE

Auto-generated code is a program source code that has been generated by program not human. There are a variety of programs that

ICPC '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3196356

```
/* Generated By:JavaCC: Do not edit this line. */
                        ...
    int ABSTRACT = 12;
    int ASSERT = 13;
    int BOOLEAN = 14;
    int BREAK = 15;
    int BYTE = 16;
                        ...
    if ((active0 & 0x1f00L) != 0L) return 16;
    if ((active0 & 0x6000L) != 0L) return 45;
    if ((active0 & 0xff00L) != 0L) return 74;
                        ...
```

Figure 1: An example of auto-generated code

generate program source code. For example, GUI builders generate parts of GUI programs and code translators transforms a program source code of a programming language to a program of another programming language. In this research, as a first step, our target is code generated by compiler-compilers.

To grasp features of compiler-compiler code, we manually checked hundreds of compiler-compiler code. As a result, we found the following features in compiler-compiler code.

- There are contiguous variable declaration statements.
- Similar conditional blocks exist contiguously.

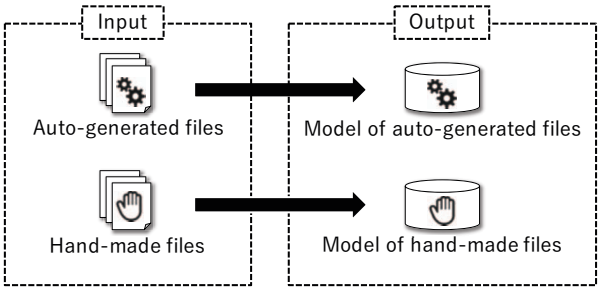
Figure 1 shows code generated by JavaCC, which is one of popular Java compiler-compiler. As shown in this figure, it is quite easy to identify auto-generated code if we see it. However, it is unrealistic to see each of source files because a project can consists of thousands of or more source files.

Auto-generated code ordinarily includes code comments that represent it was generated automatically. The source code of Figure 1 includes such a comment in the first line. We can identify auto-generated code automatically if we perform string search. However, such comments are occasionally deleted in the process of development. In the case where comments were deleted, it is impossible to identify auto-generated code by string search. Even if comments exists in auto-generated code, it is necessary to use appropriate keywords for string search.

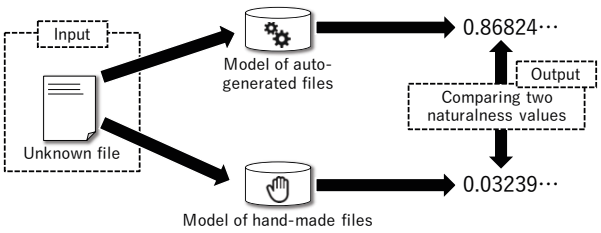
2.1 Auto-generated code identification with machine-learning techniques

Shimonaka et al. proposed a technique to identify auto-generated code automatically [9]. Their technique makes a vector from each of target source files. Every element in the vector means the number of AST nodes of the specified types except comment nodes. Their vector includes 81 node types for source files of Java version 1.6. Then, a learning model is created from vectors of known hand-made and auto-generated source files. If an unknown source file is given to the model, the model decides whether the given source file is auto-generated code or not. In Shimonaka's technique, four kinds of machine-learning algorithms are used: decision tree, naive bayes, random forest, and support vector machine.

They made a golden set of hand-made and auto-generated source files and they evaluated their technique. In many cases, their technique was able to identify auto-generated code with over 93% precision/recall. However, in some cases, prevision/recall was less than 90%. These results show that new techniques are necessary to identify auto-generated code more accurately.



(a) STEP-1



(b) STEP-2

Figure 2: Overview of the proposed technique

2.2 Key idea

After checking hundreds of compiler-compiler code, we came up an idea. The sequences of tokens in auto-generated code may have different characteristics from ones in hand-made code. That may be why human can identify auto-generated code at a glance. Consequently, we decided to use n-gram language model [1], which was developed for research on natural language processing. Recently, n-gram language model has been used in research of software engineering, too [5, 8].

N-gram language model is defined with the following formulae.

$$P(W) = \prod_{i=1}^{|W|+1} P(w_i|w_0 \ldots w_{i-1}) \quad (1)$$

$$P(w_i|w_0 \ldots w_{i-1}) = \frac{c(w_{i-n+1} \ldots w_i)}{c(w_{i-n+1} \ldots w_{i-1})} \quad (2)$$

W is a sequence of target text data. w_i is the i-th word in W. $c(w_x \ldots w_y)$ means number of occurrences of subsequence $w_x \ldots w_y$. In this research, the text data is a sequence of tokens extracted from a source file.

3 PROPOSED TECHNIQUE

Herein, we explain the proposed technique. Figure 2 shows an overview of the proposed technique. The proposed technique requires a set of known hand-made and auto-generated code in addition to unknown source file. The proposed technique decides whether the unknown file is auto-generated code or not.

The proposed technique consists of two steps.

STEP-1: the proposed technique parses all the given known hand-made and auto-generated files. The parsing results for each file is a sequence of tokens. Then every n-grams in the sequence are used to construct n-gram language models.

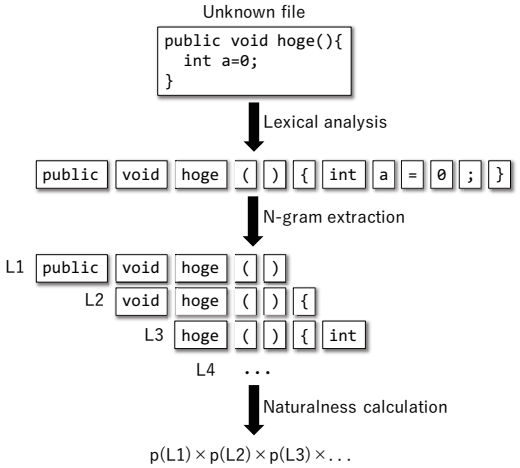


Figure 3: Calculation naturalness from unknown file

Two n-gram language models are constructed from n-grams in hand-made files and auto-generated files, respectively.

STEP-2: the given unknown file is input to the two models. Each of the both models outputs a scalar value (Figure 3). The values are naturalness as hand-made code and auto-generated code. If the value of naturalness as hand-made code is higher than the one as auto-generated code, the unknown file is regarded as hand-made file. If not, it is regarded as auto-generated code.

At present, we are using KenLM [3] for constructing N-gram language model and measuring naturalness values. If an n-gram in the test data is not included in the learning data, its naturalness becomes 0. This is a common issue in n-gram language models. To avoid this issue, KenLM uses *modified Kneser-Ney smoothing*, which is better than other smoothing techniques [4].

4 EVALUATION

Herein, we report out experimental results about the accuracy of the proposed technique.

We used Shimonaka's golden set [9] as our targets. The golden set consists of 5,000 Java source files: 1,000 hand-made, 1,000 ANTLR, 1,000 JavaCC, 1,000 JFlex, and 1,000 SableCC files. All the files except hand-made files are auto-generated code. All of them were extracted from open source projects and each of them was checked manually to avoid wrong categorization.

We conducted two experiments with the dataset.

EXP-1: measuring precision and recall with leave-one-out method. This evaluation is for comparing our technique to Shimonaka's one because they evaluated their technique with leave-one-out method. We were not able to run their technique but we were able to utilize their results for comparison.

EXP-2: measuring precision and recall with bootstrapping method. This evaluation is for measuring precision and recall of our technique more rigorously.

4.1 EXP-1

As an example, we explain the procedure in the case where we use the 1,000 ANTLR files and the 1,000 hand-made files. Firstly, we divided the 1,000 ANTLR files and the 1,000 hand-made files into 10 groups randomly. Then, we selected a group for testing data while an ANTLR model and a hand-made model were constructed from the ANTLR files and the hand-made files in the remaining 9 groups. We input each file in the selected group to both the models to calculate naturalness values. Based on the magnitude relation of the two values, we can see whether our technique works well or not. After inputting all the files in the selected group, we calculated precision and recall. This processing was repeated 10 times because we selected a different group as input files. Finally, we measured average values of precision and recall. This procedure is completely the same as Shimonaka's experiment [9].

Table 1 shows results of our technique and Shimonaka's one. In Shimonaka's technique, four kinds of machine-learning algorithm are used. "MIX" in the rightmost column in the table means models were constructed from 1,000 files that had been randomly selected from the 4,000 auto-generated files. We can see that Shimonaka's technique with decision tree has good precision and recall. In many cases, both precision and recall were over 93%. However, surprisingly, our technique score at least 98.7% precision and recall for all the cases even including MIX case. These results clearly show that

Table 1: Precision and recall of our technique and Shimonaka's one derived from leave-one-out method

Auto-generated code		ANTLR		JavaCC		JFlex		SableCC		MIX	
		Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
Our technique		100.0%	100.0%	99.6%	99.3%	99.8%	99.7%	100.0%	99.9%	98.7%	99.7%
Shimonaka's technique [9]	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 2: Precision and recall of our technique derived from bootstrapping method

Auto-generated code	ANTLR		JavaCC		JFlex		SableCC		MIX	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
Our technique	99.99%	99.98%	99.86%	99.75%	99.97%	99.86%	100.0%	99.92%	99.42%	99.84%

```
                    ...
Object visit(AndImpl node, Object data)
    throws Exception;

Object visit(BindVariableValueImpl node, Object data)
    throws Exception;

Object visit(ChildNodeImpl node, Object data)
    throws Exception;

Object visit(ColumnImpl node, Object data)
    throws Exception;
                    ...
```

(a) Similar-signature methods

```
                    ...
case '¥b': sb.append("¥¥b"); break;
case '¥f': sb.append("¥¥f"); break;
case '¥n': sb.append("¥¥n"); break;
case '¥r': sb.append("¥¥r"); break;
                    ...
```

(b) Many case entries

Figure 4: Two examples that our technique misjudged

our technique can identify auto-generated files more accurately than Shimonaka's technique.

4.2 EXP-2

In the bootstrapping method [2], N datasets are created from the original dataset. Each created dataset includes the same number of files as the original dataset but it can include the same files redundantly. Accurate estimation is performed with N datasets. Recently, bootstrapping method started to be used instead of leave-one-out method because leave-one-out tends to score higher accuracy than real. In this evaluation, we used 2,000 as N because it is said that 50 $\sim$ 2,000 are appropriate values as N.

Table 2 shows accuracy of our technique. Our technique scored over 99% precision and 99% recall for all the cases.

5 DISCUSSION

From the results of EXP-1 and EXP-2, we confirmed that our technique can identify auto-generated code with very high accuracy. However, there were some source files that had been wrongly judged by our technique. We see all of the misjudged files and found that they had the following characteristics.

- There are many method declarations whose signatures are similar to one another.
- There are many case entries.

Figure 4 shows two examples of the misjudged files. Those characteristics are common to auto-generated files, which is why our technique wrongly judged them. Shimonaka's technique tended to misjudge small files [9] while our technique worked well regardless of size of target files.

In both EXP-1 and EXP-2, precision and recall of our technique on MIX dataset tend to be less than ANTLR, JavaCC, JFlex, and SableCC datasets. This is because unique characteristics in each kind of auto-generated files become less prominent by mixing them.

6 CONCLUSION

In this paper, we proposed to use n-gram language model to identify auto-generated code automatically. The proposed technique can identify auto-generated code regardless of whether auto-generated code retains code comment such as "*generated by*" or not. In the proposed technique, two models are constructed from known hand-made code and known auto-generated code, respectively. Then, a target unknown file is given to the models. Each model measure its naturalness as hand-made code and auto-generated code, respectively. If the naturalness value as auto-generated code is higher than the one as hand-made code, it is regarded as auto-generated code.

We have evaluated the proposed technique with four kinds of auto-generated code that had been generated by compiler-compilers. In the evaluation, we compared the proposed technique with the existing one [9] and we confirmed that the accuracy of the proposed technique was higher than the existing one. More concretely, precision of the proposed technique was 98.7% or more and recall was 99.3% or more.

However, this research is still in the early stage. We have found that some hand-made files were wrongly judged as auto-generated files. We are going to enhance the proposed technique for such files. Besides, currently we have evaluated the proposed technique on only auto-generate code of compiler-compilers written by Java. In software systems, there are a variety of auto generated code. We are going to apply the proposed technique to other kinds of auto-generated code such as GUI builders code, translators and code written by other program language than Java. We also have a plan to develop an available tool of the proposed technique for researchers and practitioners.

ACKNOWLEDGMENTS

This work was supported by MEXT/JSPS KAKENHI 25220003 and 17H01725.

REFERENCES

[1] Peter F. Brown, Peter V. deSouza, Robert L. Mercer, Vincent J. Della Pietra, and Jenifer C. Lai. 1992. Class-based N-gram Models of Natural Language. *Computational Linguistics* 18, 4 (1992), 467–479.
[2] Bradley Efron. 1992. Bootstrap methods: another look at the jackknife. In *Breakthroughs in statistics*. Springer, 569–593.
[3] Kenneth Heafield. 2011. KenLM: Faster and Smaller Language Model Queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*. 187–197.
[4] Kenneth Heafield, Ivan Pouzyrevsky, Jonathan H Clark, and Philipp Koehn. 2013. Scalable Modified Kneser-Ney Language Model Estimation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*. 690–696.
[5] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering*. 837–847.
[6] Alexander C MacLean, Landon J Pratt, Jonathan L Krein, and Charles D Knutson. 2010. Trends That Affect Temporal Analysis Using SourceForge Data. In *Proceedings of the 5th International Workshop on Public Data about Software Development*. 6–11.
[7] Takafumi Ohta, Hiroaki Murakami, Hiroshi Igaki, Yoshiki Higo, and Shinji Kusumoto. 2015. Source Code Reuse Evaluation by Using Real/Potential Copy and Paste. In *Proceedings of the 9th International Workshop on Software Clones*. 33–39.
[8] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "Naturalness" of Buggy Code. In *Proceedings of the 38th International Conference on Software Engineering*. 428–439.
[9] Kento Shimonaka, Soichi Sumi, Yoshiki Higo, and Shinji Kusumoto. 2016. Identifying Auto-Generated Code by Using Machine Learning Techniques. In *Proceedings of the 7th International Workshop on Empirical Software Engineering in Practice*. 18–23.