Kode_Stylers: Author Identification through Naturalness of Code: An Ensemble Approach

Panyawut Sriiesaranusorn[a], Supatsara Wattanakriengkrai[a], Teyon Son[a], Takeru Tanaka[a], Christopher Wiraatmaja[a], Takashi Ishio[a] and Raula Gaikovina Kula[a]

[a]*Nara Institute of Science and Technology, Nara, Japan*

Abstract

Authorship identification plays an important role in detecting undesirable deception of others' content misuse or exposing the owners of some anonymous hurtful content. The Authorship Identification of SOurce COde (AI-SOCO) competition was held to investigate this task. Our team, namely *Kode_Stylers*, participated in the competition and used the naturalness of code as the key to our solution. In this working note, we (i) present methods to obtain features such as tokenization, N-gram TF-IDF, warning messages, and coding styles, (ii) implement our framework using Random Forest and Transformer to classify authors through our features, and (iii) apply an ensemble approach to increase the performance of our solutions. The results suggest that the authorship can be identified through the features extracted from source code and selected classifiers with up to an accuracy of 0.82, while the ensemble model outperforms any single model.

Keywords

Authorship Identification, Code Naturalness, Ensemble Models

1. Introduction

Authorship identification is essential to the detection of undesirable deception of others' content misuse or exposing the owners of some anonymous hurtful content. The detection facilitates solving issues related to cheating in academic, work, and open source environments. Also, it could be useful in detecting the authors of malware software over the world. This working note is in response to the call for general authorship identification of source code and is part of the Authorship Identification of SOurce COde (AI-SOCO) competition[1].

Our team, namely *Kode_Stylers*, use the naturalness of code as our key intuition behind the author identification. We depict code as being repetitive and predictable when compared to English, much due to the presence of *language specific syntax* patterns such as separators, operators, and keywords. Modelling programming languages as a language has made popular the term 'natural software', which is now commonly used as a means to understand how code is written. The naturalness of software refers to the repetitive nature of the code in a project [2]. Language modelling has revealed power-law distributions and the naturalness of source code [3]. Natural software is useful for understanding refactoring activities [4, 5, 6], contributions [7] , finding buggy code [4, 8, 9], and so on.

Forum for Information Retrieval Evaluation, December 16-20, 2020, Hyderabad, India
EMAIL: sri-iesaranusorn.panyawut@is.naist.jp (P. Sriiesaranusorn)

2. Methodology Used

In this section, we present the dataset and explain how we treat source code as a natural language model. Additionally, we discuss in the detail the different materials and methods used in our submission.

2.1. Dataset

The dataset consists of source code files and their corresponding authors collect from an open submission in the Codeforces. All source code files are implemented in C++ programming language, contain comments, and are bug-free. More specifically, there are the selected 1,000 authors and 100 collected source code files per one author, a total of 100,000 source code files. The dataset is divided into three parts, 50% of the training set, 25% of the validating set, and 25% of the testing set, ensuring approximately equal ratios among authors.

2.2. Similarity of Tokenized Source Code

Treating source code as a raw language, we perform tokenization which breaks a stream of text into words called tokens [10]. Considering the programming syntax, we select the NCDSearch tool implemented by [11] to do word-level tokenization. This tool is based on grammar from the lexer files generated by the ANTLR4 parser generator [12]. It allows us to feed source code as input, and return the word-level tokens as output. Along with the tokenization process, NCDSearch also removes source code comments from source code.

Once we obtain the word tokens, preprocessing is required to handle low-frequency words or rare word problems. Due to the different programming styles, one of the rare word types is the variable name used in each source code file. According to a previous work [13], this problem can be a cause of the inefficiency of classification models, which needs to be solved. Based on the distribution of our data, we mask the low-frequency words as UNK token if the occurrence of words is less than 10 in the dataset. With the limitation of our computational resource, we select the first 3,000 tokens per each file to train the model, because most of our data contain tokens less than 3,000 per each file. On the other hand, we perform padding in the case that the total number of tokens is less than 3,000 tokens.

2.3. Similarity of N-gram TF-IDF in Source Code

Regarding the similarity of source code and raw language, we apply N-gram IDF [14, 15] to extract the n-gram features. Inverse Document Frequency (IDF) measures how important a term is; however, IDF cannot measure the importance of phrases, i.e., multi-word terms. More specifically, some useful phrases are assigned less weight than rare phrases since IDF gives more weight to rare terms which are found in fewer documents. N-gram IDF, a theoretical extension of IDF, can be used to handle multiple terms and phrases by connecting the term weighting and multi-word expression extraction [14, 15]. N-gram IDF has been widely used in the field of text classification for bug reports [16] and self-admitted technical debt (SATD) [17, 18].

To enumerate valid n-gram terms, we use an n-gram weighting scheme tool [15] which uses an enhanced suffix array [19]. Since the tool removes all special characters (e.g. ;) and

Table 1
Compiler options and description[20, 21, 22]

options	Description
-Wall	Enables commonly used warning options pertaining to usage that we recommend avoiding and that we believe are easy to avoid.
-Wextra	Enables some warning options for usages of language features which may be problematic.
-Wc++compat	Warn about ISO C constructs that are outside of the common subset of ISO C and ISO C++
-Wc++14-compat	Warn about C++ constructs whose meaning differs between ISO C++ 2011 and ISO C++ 2014
-Wconversion	Warn for implicit conversions that may alter a value.
-Wdate-time	Warn when macros __TIME__, __DATE__ or __TIMESTAMP__ are encountered as they might prevent bit-wise-identical reproducible compilations.
-Wdisabled-optimization	Warn if a requested optimization pass is disabled.
-Wlogical-op	Warn about suspicious uses of logical operators in expressions.
-Wold-style-cast	Warn if an old-style (C-style) cast to a non-void type is used within a C++ program.
-Wunused-macros	Warn about macros defined in the main file that are unused.
-Weffc++	Warn about violations of the following style guidelines from Scott Meyers' Effective C++series of books.
-Wfloat-equal	Warn if floating-point values are used in equality comparisons.
-Wpedantic	Issue all the warnings demanded by strict ISO C and ISO C++.
-Wshadow	Warn whenever a local variable or type declaration shadows another variable, parameter, type, class member (in C++)
-Wvariadic-macros	Warn if variadic macros are used in ISO C90 mode, or if the GNU alternate syntax is used in ISO C99 mode

ignores capital characters during its process, we encode such special characters with terms (e.g. `semicolon`) and then do lowercase on all terms before applying the tool. The output of the tool is a list of all valid n-gram terms with a maximum length of 10. We obtain more than one million n-gram terms from all source code in the training set. We calculate N-gram TF-IDF scores, a measure of how significant an n-gram term is, of all n-gram terms to find n-gram terms that would be useful in author identification. The higher the N-gram TF-IDF score, the more important the n-gram term is. The score is defined as:

$$N\text{-}gram\ TF\text{-}IDF = log(\frac{|D|}{sdf}) * gtf$$

Where $|D|$ is the total number of source code files (in the training set), sdf is the document frequency of a set of words composing n-gram, and gtf is the global term frequency of the term. We use the top five percent of n-gram terms which have the highest score as features for classifier learning. For each source code file, we create a feature vector of frequencies that n-gram terms appear in the file. The intuition behind this approach is that authors tend to use repetitive terms in their code when programming.

2.4. Similarity of warning messages generated by Source Code

We assume that the compiler warnings are useful to extract the coding styles or naturalness of source code authors. Compiler warnings are used to detect bugs by matching source code with certain patterns which are likely to be causes of programming errors[23]. In programming competitions, coding speed is one of the important factors to win the coding contests. Some participants tend to use the same code snippets for solving resembling problems. Furthermore, they intentionally ignore the manners of C++ to boost their coding speed (e.g. use “int” instead of “size_t”).

To find the programmer coding style, we compile all the source code with the following conditions and then extract compiler warnings. We use *gcc* compiler [24] with 5 types of C++ versions (C++98,C++03,C++11,C++14,C++17), and 15 options. Table 1 shows the compiler options description. These compiler options point out not only deprecated behaviors but also coding styles. For example, options -Wold-style-cast can detect the old-style casting[20].

2.5. Similarity of Naming Conventions (Identifiers) in Source Code

We assume that the identifiers of the programs could provide useful information to identify the authors of the source code. Identifiers are the tokens used to uniquely identify program elements, such as variables and functions, in source code. Approximately 70% of the source code of a software system consists of the identifiers[25], and some of the identifiers are user-defined names.

To extract identifiers from source code, we use the Code Hash Tool[26] which implements the ANTLR4-based lexical analyzer. This tool extracts a token sequence by removing comments and white spaces, and then returns a list of word-level tokens, along with information on whether the tokens are identifiers, in JSON format. Afterwards, we extract the identifiers such as the variable name, and the non-identifiers (e.g., brackets) from the source code.

3. Implementation Techniques

In section, we present the implementation of our framework as the following pipeline.

3.1. Transformer Model

To build the classification models, we select the Transformer model which is used primarily in the field of natural language processing (NLP) [27]. Previous studies used this model and achieved outstanding performance, compared to the other deep neural networks and traditional machine learnings [28, 29]. The transformer in this study is adopted from the source code in [30]. The architecture starts from embedding layers with 128 embedding size, followed by a transformer layer with 4 multi-head and 64 hidden neurons. We then feed the extracted features to a simple feed-forward model, consisting of 2 fully-connected hidden layers, each regularized by batch normalization and 40% dropout for regularization [31, 32]. Lastly, the output layer uses the softmax activation function to simulate a probability vector, as our task is a multi-class classification.

3.2. Random Forest Model

According to the prevalence of Random Forest (RF) in the field of natural language processing [33, 34, 35], we select it as one of the candidate models. Random Forest is an ensemble technique

Table 2
The description and individual result of five single models

	Features	Classifier	Accuracy
Model A	Word-level tokens	Transformer	0.7385
Model B	Word-level tokens and N-gram TF-IDF	Random Forest	0.8202
Model C	Word-level tokens, Non-Identifier, and N-gram TF-IDF	Random Forest	0.7452
Model D	Word-level tokens, Warning message, and N-gram TF-IDF	Random Forest	0.7848
Model E	Word-level tokens, Identifier, and N-gram TF-IDF	Random Forest	0.7501

that produces many classification and regression trees, where each tree is constructed by bootstrap samples with a subset of features. The prediction of the random forest of the ensemble is determined by combining the final decision of each tree. The advantage of this model is to deal with the problem of noise or outliers which may possibly affect the result of the overall classification method [36].

3.3. Ensemble of our proposed Models

As we have several models based on several approaches, the ensemble method is selected to merge the results. In machine learning, this method allows us to combine several models based on criteria such as majority voting or averaging, in order to produce one optimal predictive model [37]. Prior works show that an ensemble method achieves the outstanding performance, compared to using a single model [38, 39]. The ensemble used in this study is defined as:

$$Ensemble = \frac{\sum_i^N w_i M_i(x)}{\sum_i^N w_i}$$

where w_i is the selected weight for the output vector from the model $M_i(x)$, and N is the total number of selected models in this study.

4. Room for Improvements

Table 2 shows the results of the five main experiments according to the variety of features and classifiers. The selected feature and classifier of models are different. Model A is based on the technique of naturalness of source code with the hypothesis that ”authors tend to write code that has similar naturalness” in their code. Model B is based on the term-frequency technique with the hypothesis that ’authors tend to use repetitive terms in their code’ when programming. Model C is based on the technique of looking at the coding syntax patterns with the hypothesis that ’authors tend to use the same coding syntax’ when coding. Model D is based on the technique of investigating different compiler warnings with the hypothesis that ’authors tend to have the same compiler warnings’ when writing code. Model E is based on the technique of identifier analysis with the hypothesis that ’authors tend to use the same identifiers’ when writing code. All single models are evaluated on the test set of the competition. The results suggest that Model B outperforms other single models.

Table 3
The result of using ensemble with five main models

	Ensemble Threshold					Accuracy
	Model A	**Model B**	**Model C**	**Model D**	**Model E**	
Ensemble#1	0.18	0.24	0.20	0.18	0.20	0.8067
Ensemble#2	0.12	0.44	0.32	0.06	0.06	0.8090
Ensemble#3	0.50	0.50	-	-	-	0.8091
Ensemble#4	0.15	0.31	0.23	0.08	0.23	0.8126
Ensemble#5	0.10	0.45	0.45	-	-	**0.8208**

Table 3 shows the improvement of results when we use different weights through single results. The Ensemble#5, which is a combination of Model A, B, and C, is accurate more than other ensemble results, including the single Model B. These results suggest that the ensemble result achieves higher performance.

With the limited time of the competition, we can perform only five main models and some ensembles. To improve the performance of the model, we aim to investigate the other features by applying software engineering knowledge. For tokenization, examining the performance of NCDSearch and the other tools is required to improve the preprocessing. Considering the hyper-parameter tuning of random forest and transformer models, the current results possibly are not based on the best set of hyper-parameters. In the future, we plan to explore the hyper-parameters and other approaches.

5. Conclusion

This working note presents models for author identification through the various hypotheses, especially the naturalness of code. Our results suggest that the ensemble model achieves the best performance, compared to the single models. We discuss the possibility for future works of this study such as finding the other features, applying the other tokenization tools, and tuning hyper-parameters of the classifiers.

Acknowledgments

This work has been supported by the Japan Society for the Promotion of Science (JSPS) KAKENHI Grant Numbers JP18H04094, JP18KT0013, JP20K19774, and JP20H05706.

References

[1] A. Fadel, H. Musleh, I. Tuffaha, M. Al-Ayyoub, Y. Jararweh, E. Benkhelifa, P. Rosso, Overview of the PAN@FIRE 2020 task on Authorship Identification of SOurce COde (AI-SOCO), in: Proceedings of The 12th meeting of the Forum for Information Retrieval Evaluation (FIRE 2020), CEUR Workshop Proceedings, CEUR-WS.org, 2020.

[2] R. Robbes, M. Lanza, Improving code completion with program history, Automated Software Engg. 17 (2010) 181–212.

[3] M. Rahman, D. Palani, P. C. Rigby, Natural software revisited, in: Proceedings of the 41st International Conference on Software Engineering (ICSE), 2019, pp. 37–48.

[4] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, P. Devanbu, On the "naturalness" of buggy code, in: Proceedings of the 38th International Conference on Software Engineering (ICSE), 2016, pp. 428–439.

[5] R. Arima, Y. Higo, S. Kusumoto, Toward refactoring evaluation with code naturalness, in: Proceedings of the 26th Conference on Program Comprehension, 2018, p. 316–319.

[6] B. Lin, C. Nagy, G. Bavota, M. Lanza, On the impact of refactoring operations on code naturalness, in: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2019, pp. 594–598.

[7] T. Bunkerd, D. Wang, R. G. Kula, C. Ragkhitwetsagul, M. Choetkiertikul, T. Sunetnanta, T. Ishio, K. Matsumoto, How do contributors impact code naturalness? an exploratory study of 50 python projects, in: 2019 10th International Workshop on Empirical Software Engineering in Practice, 2019, pp. 7–75.

[8] E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, J. N. Amaral, Syntax and sensibility: Using language models to detect and correct syntax errors, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 311–322.

[9] H. Campbell, A. Hindle, J. Amaral, Syntax errors just aren't natural: Improving error reporting with language models, 11th Working Conference on Mining Software Repositories, MSR 2014 - Proceedings (2014).

[10] R. Renu, D. Gaur, Tokenization and filtering process in rapidminer, International Journal of Applied Information Systems 7 (2014) 16–18.

[11] T. Ishio, N. Maeda, K. Shibuya, K. Inoue, Cloned buggy code detection in practice using normalized compression distance, in: Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018, pp. 591–594.

[12] ANTLR, 2020. URL: https://github.com/antlr/antlr4.

[13] B. Heap, M. Bain, W. Wobcke, A. Krzywicki, S. Schmeidl, Word vector enrichment of low frequency words in the bag-of-words model for short text multi-class classification problems, ArXiv abs/1709.05778 (2017).

[14] M. Shirakawa, T. Hara, S. Nishio, N-gram idf: A global term weighting scheme based on information distance, 2015, pp. 960–970.

[15] M. Shirakawa, T. Hara, S. Nishio, Idf for word n-grams, ACM Trans. Inf. Syst. 36 (2017).

[16] P. Terdchanakul, H. Hata, P. Phannachitta, K. Matsumoto, Bug or not? bug report classification using n-gram idf, in: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017, pp. 534–538.

[17] S. Wattanakriengkrai, R. Maipradit, H. Hata, M. Choetkiertikul, T. Sunetnanta, K. Matsumoto, Identifying design and requirement self-admitted technical debt using n-gram idf, in: 2018 9th International Workshop on Empirical Software Engineering in Practice (IWESEP), 2018, pp. 7–12.

[18] R. Maipradit, C. Treude, H. Hata, K. Matsumoto, Wait for it: identifying "on-hold" self-admitted technical debt, Empirical Software Engineering (2020) 1 – 29.

[19] M. I. Abouelhoda, S. Kurtz, E. Ohlebusch, Replacing suffix trees with enhanced suffix arrays, Journal of Discrete Algorithms 2 (2004) 53 – 86.

[20] 3.8 Options to Request or Suppress Warnings, 2020. URL: https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html.

[21] 2.4 Options to request or suppress errors and warnings, 2020. URL: https://gcc.gnu.org/onlinedocs/gfortran/Error-and-Warning-Options.html.

[22] 3.5 Options Controlling C++ Dialect, 2020. URL: https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Dialect-Options.html.

[23] C. Sun, V. Le, Z. Su, Finding and analyzing compiler warning defects, Proceedings of the 38th International Conference on Software Engineering (ICSE) (2016) 203–213.

[24] GCC, the GNU Compiler Collection, 2020. URL: https://gcc.gnu.org/.

[25] F. Deissenbock, M. Pizka, Concise and consistent naming, Software Quality Journal 14 (2006) 261–282.

[26] CodeHash Tool, 2019. URL: https://github.com/NAIST-SE/CodeHash.

[27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, u. Kaiser, I. Polosukhin, Attention is all you need, in: Proceedings of the 31st International Conference on Neural Information Processing Systems, 2017, p. 6000–6010.

[28] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, in: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, 2019.

[29] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. Le, R. Salakhutdinov, in: Transformer-XL: Attentive Language Models beyond a Fixed-Length Context, 2019, pp. 2978–2988.

[30] Transformer, 2020. URL: https://github.com/keras-team/keras-io/blob/master/examples/nlp/text_classification_with_transformer.py.

[31] S. Ioffe, C. Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift, in: Proceedings of the 32nd International Conference on Machine Learning, volume 37, 2015, pp. 448–456.

[32] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: A simple way to prevent neural networks from overfitting, Journal of Machine Learning Research 15 (2014) 1929–1958.

[33] H. Parmar, S. Bhanderi, G. Shah, in: Sentiment Mining of Movie Reviews using Random Forest with Tuned Hyperparameters, 2014.

[34] M. Z. Islam, J. Liu, J. Li, L. Liu, W. Kang, A semantics aware random forest for text classification, in: Proceedings of the 28th ACM International Conference on Information and Knowledge Management, 2019, p. 1061–1070.

[35] K. Kowsari, K. Jafari Meimandi, M. Heidarysafa, S. Mendu, L. Barnes, D. Brown, L. Id, Barnes, Text classification algorithms: A survey, Information (Switzerland) 10 (2019).

[36] G. Shah, H. Parmar, Experimental and comparative analysis of machine learning classifiers, International Journal of Software Engineering and Knowledge Engineering 3 (2013) 9.

[37] F. Huang, G. Xie, R. Xiao, Research on ensemble learning, in: 2009 International Conference on Artificial Intelligence and Computational Intelligence, volume 3, 2009, pp. 249–252.

[38] A. Jurek, Y. Bi, S. Wu, C. Nugent, A survey of commonly used ensemble-based classification techniques, The Knowledge Engineering Review 29 (2014) 551–581.

[39] I. D. Mienye, Y. Sun, Z. Wang, An improved ensemble learning approach for the prediction of heart disease risk, Informatics in Medicine Unlocked 20 (2020) 100402.