Using Machine Learning Methods to Establish Program Authorship

Sergey Gorshkov, Maxim Nered, Eugene Ilyushin, Dmitry Namiot

***Abstract*— The subject of the article is the “coding style” concept and the main approaches to detecting the individual style of a programmer. The entire process of creating a software product from this point of view and the main features of programming style are analyzed. It emphasizes the relevance and commercial significance of the problem in terms of product support, plagiarism, work of a large developer’s community in a single repository, an evolution of developer skills. Computational stylometry issues, a possibility of using programming paradigms as an additional factor of style identification are considered. It offers the idea of creating a software tool that allows to identify the style of the author who wrote a particular program fragment and allows less experienced developers to follow the rules accepted in the major part of the repository and determined by coding style of "experts", which leads the code to a uniform format that is easier to maintain and make adjustments. Globally, this stage of analyzing the original (and then the modified code) allows improving the existing algorithms for automatic synthesis of programs.**

***Keywords*—coding style, programming paradigms, computational stylometry, plagiarism, deanonymization.**

I. Introduction

Nowadays, computer programs are developed, as a rule, cooperatively by a large number of developers. There are many services for hosting projects and their development, the largest of which is *Github*. Every developer, whether new or professional, has a unique programming style that can change over time. Often, a team leader or a product manager would like all developers to adhere to a certain programming style in the project, which would make the code more readable and improve teamwork. Requirements for the source code can also be set at the company / unit level (so-called style guide e.g. *https://google.github.io/styleguide/javaguide.html*). Now such control can be carried out, for example, by programs like "linter". However, this approach is static and does not respond to changes in the repository style, which, for example, may arise due to the prevalence of the code share written by a new team member. The second practical question considered in this article is connected with the definition of the code author. It is in demand in many areas of business activities and law enforcement. Currently, a lot of research is being carried out, solving the problem in various ways, including methods of machine learning.

The research shows that about 80% of the software life cycle cost comes from servicing the finished product due to insufficient software quality. This may be connected to the logical component and its implementation, as well as to deeper problems that may not immediately be revealed, such as code readability, documentation quality, clear traceability of relationships in software, and so on. In many cases, the only source of information about a software product is its source code and the developer who wrote it. Adherence to the coding style as one of the basic requirements for software product development helps improve the quality of the code.

Let's describe the terminology used in this area, highlight the main features of the problems under consideration and the available solutions.

II. Programming Style and Evolution of Developer Skills

Programming (coding) style is an intuitive and seemingly elusive concept which shows the style of writing code. This is a purely individual characteristic, it is easily recognized "by eye", but it is rather difficult to make an assay of the problem. The goal of adhering to programming style is to make the program understandable, which makes it easy to work with, but individual programming style that is different from that of other team members often deteriorates the readability and understanding of the source code. Obviously, writing comments, use of meaningful names of code elements and satisfying the basic requirements for writing code in the language used, which are embedded in many modern IDEs, can be attributed to the “good” coding style.

The concept of a coding standard differs from the concept of programming style: the former is a set of practices recognized to be successful in the sector that involve many recommendations for the development of programming code. There are studies confirming that adherence to coding standards in software development can improve teamwork, reduce errors in a software product, and improve code quality. Working in accordance with coding standards, team members understand their colleagues' programs more easily and eliminate errors in them [1]. Fundamentally, standards for writing code are nothing more than the evolution of programming styles. When a programming style becomes

Manuscript received 05.12.2018
Sergey Gorshkov is with Lomonosov Moscow State University (e-mail: serggorsar@yandex.ru)
Maxim Nered is with Lomonosov Moscow State University (e-mail: freepvps@gmail.com)
Eugene Ilyushin is with Lomonosov Moscow State University (e-mail: eugene.ilyushin@gmail.com).
Dmitry Namiot is with Lomonosov Moscow State University (e-mail: dnamiot@gmail.com).

popular and gains public acceptance, it rises to coding standards.

An important point is an observation that the programming style is closely related to a developer's skills progress. As the level grows from beginner to lead developer, the quality of the code written by him will increase. This evolution can be divided into three stages.

Stage 1. These are, as a rule, newcomers to the profession, who do not know acknowledged coding standards and do not have their own style, they write programs according to their idea of style, so source code written by them is often illogical and has low readability.

Stage 2. Developers have an individual programming style based mainly on coding standards.

Stage 3. Lead developers know the coding standards, have their own ideas about a style of writing code, can use their own insights, for example, for working with files, handling database connections, managing virtual memory, etc., to make the programs more efficient, reliable and portable from their point of view. Such divergence, in this case, we can consider as amendments to the standards, and it is necessary to distinguish such anomalies from beginners' failures in writing code.

A programming style can be defined as an interpretation by a programmer or a company [2] of a set of rules and their use for writing source code to achieve the goal. A set of rules applicable to writing source code can be divided into four main areas [3]:

1. General programming practices - rules and recommendations regarding methodology and language that affect the source code.
2. Typographic styles - rules that affect only the layout of the source code and the use of comments, but not the execution of the program.
3. Control structure styles - rules that affect the use of algorithms and their implementation, and control constructs.
4. Information structure styles - rules affecting data structure, flows, data storage and operations.

III. Stylometry and Authorship Identification of Source Code

Stylometry is a statistical analysis of style that complements the traditional methods of literary analysis. Stylometry usually includes studies that use style as an indicator, for example, the author's stylistic peculiarities as proof of his authorship or certain changes in the author's style as an indicator of the works chronology. Equally important are also statistics that are purely descriptive. An overwhelmingly important feature of the programming style is that the style is unique to a person, like his fingerprint or retina. This is called the hypothesis of human styloma, which suggests that authors can be distinguished by measuring the specific properties of their works, called stylomas [4, 5].

A perspective area of stylometry is computational stylometry. It describes and explains the cause-and-effect relationship between the psychological and social properties of the authors, on the one hand, and their style of writing, on the other. The results of studies in this field of science can be used to develop systems that generate text in a particular style, or systems that recognize the identity of the authors or some of their personal traits, using the text written by them. This field will be considered further.

Computational stylometry is used within natural language processing (NLP) tasks as one of three text comprehension levels. The purpose of text comprehension is to extract knowledge from the text and to present it in a format that is reusable. Over the past decade, NLP has made significant progress by switching to statistical and machine learning methods in research and increased interest due to commercial applicability (Apple's SIRI, Yandex's Alice, Amazon's Alexa, Google assistant are examples of recent most advanced commercial NLP applications). Three categories of knowledge that can be extracted from text [4] are as follows:

1. Objective knowledge (answer to special questions: who, what, where, when, ...)
2. Subjective knowledge (who has what opinion and to what extent)
3. Some metadata (what we can extract from the text separately from its content, mainly about its author).
1. Computational stylometry solves issues falling in the last category. We describe a set of basic tasks solved by the methods described.

Firstly, it is the programmer's deanonymization task. It is statement will be as follows - some analyst is interested in the identification of an anonymous programmer. This could be a privacy concern for open source authors who want to remain anonymous.

Secondly, it is the detection of ghostwriting (a situation in which an author writes texts, or in our case, programs, for another person, and at the same time his authorship is not mentioned anywhere). Ghostwriting detection is associated with traditional plagiarism detection. There are many ready-made commercial solutions to this issue, such as MOSS ([22]), JPlag ([23]) and Sherlock ([24]).

Thirdly, it is legal expertise of software. In this case, an analyst collects many candidate programmers on the basis of previously obtained malware samples or code repositories.

Fourthly, it is copyright study. Borrowing code often leads to copyright disputes. Informal mechanisms for hiring programmers are widespread, and in the absence of a written contract, someone may require to be acknowledged as the author of a part of the code after it has been written for hire and sent.

An example of a finished software product is the Smart Formatter [6], which analyzes the source code quality from three different points of view: indentation style, naming style and use of comments and their frequency. Indentation style is studied by analyzing for each grammar rule the relative position of each terminal or nonterminal that makes up a rule relative to the previous token. Indentation rule is obtained by applying descriptive statistics (average or median) on the collected positions for each instance of the grammar rule. To process comments, the tool analyzes their frequency and extracts source code files with a comment frequency below a predetermined threshold.

IV. Basic approaches to problem solving

Many software products that solve problems associated with the style of writing code are based on the use of various methods of machine learning. Traditional methodology for obtaining software for the task, used in this area, usually involves the following steps:

1. Extracting software metrics that could define an author's style
2. Filtering metrics and highlighting the really significant ones
3. Choosing a machine learning model for classifying and training the model using selected metrics
4. The application of the model is based on the selection of an already filtered set of metrics.

In most studies, priority is given either to the first stage, related to the choice of metrics (steps 1 and 2), or to the second one (step 3), which is related to the model choice. This paper focuses on the first step.

In order to apply the machine learning methods, it is necessary to distinguish a set of features - characteristics (explanatory variables, attributes) of the source code, for which we assume that they can affect the identification of the author. Of course, it is necessary to consider many factors, to be able to evaluate their contribution, pairwise correlation, the problem of retraining. We can identify the main directions of the characteristics search as follows:

1. Lexical metrics. Source code analysis by highlighting metrics associated with lexical features: keywords of a language, functions, macros, comments, preprocessor directives, etc. The occurrence frequency of these structures, their average, and total length, the number of unique elements, etc. can be considered as the metrics. In addition to the general selection of entities, their features, for example, their type can be used. Thus, the text is converted into a sequence of primitive objects - lexical tokens. On the ground of these objects, we can build a model that can take into account the frequency of encountered tokens, their context, correlations, etc. This is used, for example, in the papers [7, 8].
2. Layout metrics and style metrics. The general layout and characteristics of syntactic constructions are analyzed as well as the use of such language elements as spaces and tabs, naming style of variables, etc. For example, it may be the distribution of lengths and numbers, as well as their standard deviations, strings, characters, numeric literals, standardized to the characteristics of strings and files; metrics related to the number of leading spaces, the use of spaces / tabs, underscores, semicolons, commas, etc. This is used, for example, in the papers [9, 10].
3. Styles of the control and information structure - the use of various algorithms and their implementation, control constructs, data structures. For example, organizing cyclical methods, using classes / functions, branching, using equivalent tools for algorithm implementation. This especially affects languages with a large amount of syntactic sugar, for example, Perl with its "There's more than one way to do it". Differences in the use of basic concepts - records (data structures: groups of references to data elements with indexed access to each element), lexically closed closures, independence (sequential / parallel), and named states are also a programmer's distinctive feature, inspired by a programming language, but however, providing considerable variation. This is used, for example, in [11].
4. For analyzing the structure, an abstract syntactic tree is often used, which is an intermediate representation of the program between a parse tree and actual data structure, which makes it easier to distinguish such features as location of control structures, loops, nesting levels of operators and operands of various types, branching, number of function parameters and other. This is used, for example, in [12, 13, 14].
5. N-gram analysis of the original text or bytecode. In this approach, n consecutive elements are analyzed as well as the occurrence frequency of these sequences. As a rule, a certain number of most frequently encountered n-grams is allocated for each author, and when analyzing programs, the overlap size of the most frequently encountered n-grams sets for the candidates and the program are evaluated. Context analysis of each of the n-grams can also be used. This is used, for example, in [15, 16].
6. Project architecture is decomposition of a system into its implementation modules and dependencies between them. The indicators obtained from revision history, as a result of which the writing style of various parts of a project was mixed, will be quite effective and useful. In papers [17] and [18], it was shown that the number of change metrics are important for files, where joint changes were carried out in one and several architectural modules. Suddenly, the metric of the number of strings in a file and in the implementation components - functions, classes, etc. - becomes one of the most significant.
7. Special attention is to be paid to a rather unexplored issue, which looks very promising due to its global nature. It is programming paradigms where you can observe how a programmer uses certain concepts - basic elements in a given hierarchy. Each concept implements a certain language functional, and a set of concepts defines a common paradigm. For example, discrete synchronous programming is best for reactive problems, i.e. problems which consist of reactions to sequences of external events. A proper understanding of concepts can help improve a programming style even in languages that do not directly support them, just like object-oriented programming is possible in C language with correct programmer's attitude. Program states are very important - they can be named and unnamed, deterministic and non-deterministic, and sequential

or parallel. The least expressive combination is the functional programming monads, the most expressive is object-oriented programming with support of messages exchange and shared resources. Many languages support two or even three paradigms. The first paradigm is chosen for the problem that is most often language-oriented. The second paradigm is chosen to support abstraction and modularity and is used when writing large programs. There can be three complementary paradigms, for example, in SQL: relational programming mechanism for logical queries to databases and transaction interface for parallel database updates + host language that supports OOP. How much a programmer uses the concepts of each of these paradigms will certainly determine his style. For example in relation to SQL, there are dialects with code inserts in other programming languages, there are multiple tools based on different aspects, which allow writing code with methods and extensions of the language itself.

Many groups of metrics (especially 2 and 3) depend on a programming language, therefore they cannot be recognized as universal. In addition, many programming features are subject to style agreements, for example, PEP8 in Python, embedded in many IDEs.

As a result of metrics allocation in various ways, there may be too many indicators that need to be filtered. Their selection is a nontrivial process and usually involves setting thresholds to eliminate the indicators that have little effect on the classification model. Usually, this happens as follows: the metrics inherent to a relatively small group of authors are selected, considering that they define their style and are not inherent to other programmers. For this, Shannon informational entropy is often used (a measure of uncertainty for the metric and the author). Individual consistency is calculated - that is, how randomly the code determines the metrics for programs of a particular developer, and then population consistency - for all developers' programs. After that, we minimize the ratio of individual entropy to the group one, because low entropy indicates that there are very few cases of using this metric and it is quite unique for developers. Thus, there is a selection of metrics and their top (a certain number of metrics for which the entropy-based figure is minimal) is used as features for further learning.

It is beyond argument that in studies not one type of features is usually used, but several, and, as a rule, it is their nature that will determine the machine learning method that will be used to solve an application task. In studies, for example, the method of support vectors ([11]), the Bayesian classifier ([19]), neural networks ([20]), and various combinations are common.

The accuracy of the classification depends on the set of selected characteristics and the method of machine learning. The papers show that with the increase in the number of candidates, the prediction accuracy decreases. Let us take as an example the results of some studies, where testing is carried out on the programs of the same authors, the model

Study	Feature selection method	Machine learning method	Number of authors	Accuracy
[16]	n-grams	–	30	97%
[19]	Lexical/ layout/ style	Voting Feature Interval	12	76%
[20]	AST	Random Forest	70	73%
[20]	AST	Neural network	70	89%
[21]	AST+ Lexical/ layout/ style	Neural network	250	98%

is trained on:

Tab. 1. Training methods and classification accuracy

V. Further research

In future, the work is planned on methods of style identifying with the help of tokenization and the use of various machine learning methods. Undoubtedly, there are still a lot of open issues in programming style study and identification of code authors. In the following steps, it is intended to explore the possibility of using the taxonomy of programming paradigms to classify source code and determine the style of a whole project. It is also planned to use combinations of other machine learning methods for the target task, to consider more complex composite metrics. It is planned to develop a product that would work for almost any source text different only at the tokenization stage, be it a literary text or a program.

Another possible trend is connected with detailed study of using the model for classifying code in various programming languages and use of syntax features of these languages. An interesting area of research is the portability of the style of one developer to different programming languages, both related and using fundamentally different paradigms with tokenization dependent on the language syntax.

VI. Conclusion

In this article, an overview of the areas in which the key role is played by programming style was given, computational stylometry and coding standards were considered, as well as their characteristics and correlation with the style. The connection of the programming style and work with large repositories was considered, the evolution of developers' skills and general requirements for a particular style were highlighted. The traditional approach in tasks of this type to determining the authorship of a program was considered - the steps which form a tool for this task were described. Various approaches to allocation of features for machine learning were discussed in detail: these are lexical metrics, location / style metrics, control and information structure style metrics, use of abstract syntax tree, n-gram analysis, project architecture research and use of programming paradigms. The results of some studies using a different set of characteristics and machine learning models were also presented. The final section presents areas of future research.

References

[1] Y. Wang, B. Zheng, H. Huang. “Complying with Coding Standards or Retaining Programming Style” // Journal of Software Engineering and Applications, pp. 1:88-91, 2008

[2] A. Mohan, N. Gold. “Programming Style Changes in Evolving Source Code” // IEEE, 2004

[3] P. Oman, C. Cook. “A taxonomy for programming style” // 18th ACM Computer Science Conference Proceedings, pp. 244-247, 1990

[4] D. I. Holmes. “Stylometry” // Encyclopedia of Statistical Sciences, 2006

[5] W. Daelemans. “Explanation in Computational Stylometry” // Springer: International Conference on Intelligent Text Processing and Computational Linguistics, pp 451-462, 2013

[6] F. Corbo, C. Del Grosso, M. Di Penta. “Smart Formatter: Learning Coding Style from Existing Source Code” // Software Maintenance. IEEE International Conf. 2007. Pp. 525-526.

[7] H. Ding, M. Samadzadeh. “Extraction of java program fingerprints for software authorship identification” // Journal of Systems and Software 72, 1 (2004), 49–57.

[8] J. Hayes, J. Offutt. “Recognizing authors: an examination of the consistent programmer hypothesis” // Journal of Software Testing, Verification and Reliability 20, 4 (2010), 329–356.

[9] A. Gray, P. Sallis, S. MacDonell. “Software forensics: Extending authorship analysis techniques to computer programs” // Information Science Discussion Papers Series No. 97/14

[10] E. Spafford, S. Weeber. “Software forensics: Can we track code to its authors?” // Computers & Security 12, 6 (1993), 585–595.

[11] B. Pellin. “Using classification techniques to determine source code authorship”. // White Paper: Department of Computer Science, University of Wisconsin (2000).

[12] D. Yu, X. Peng, W. Zhao. “Automatic refactoring method of cloned code using abstract syntax tree and static analysis” // Journal of Chinese Computer Systems 30(9), 1752–1760 (2009)

[13] I. Baxter, A. Yahin, L. Moura, M. Sant’Anna, L. Bier. “Clone detection using abstract syntax trees”. // Software Maintenance, 1998. Proceedings., International Conference on. pp. 368–377. IEEE (1998)

[14] F. Lazar, O. Banias. “Clone detection algorithm based on the abstract syntax tree approach” // Applied Computational Intelligence and Informatics (SACI), 2014 IEEE 9th International Symposium on. pp. 73–78. IEEE (2014)

[15] G. Frantzeskou, S. MacDonell, E. Stamatatos, S. Gritzalis. “Examining the significance of high-level programming features in source code author classification”. // Journal of Systems and Software 81, 3 (2008), 447–460.

[16] G. Frantzeskou, E. Stamatatos, S. Gritzalis, S. Katsikas. “Effective identification of source code authors using byte-level information” // Proceedings of the 28th International Conference on Software Engineering (2006), ACM, pp. 893–896.

[17] E. Kouroshfar, M. Mirakhorli, H. Bagheri, L. Xiao, S. Malek and Y. Cai. “A Study on the Role of Software Architecture in the Evolution and Quality of Software”. // Proceedings of the 12th Working Conference on Mining Software Repositories, (2015) 246-257

[18] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo. “A review-based comparative study of bad smell detection tools” // Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, ACM, 2016, p. 18.

[19] J. Kothari, M. Shevertalov, E. Stehle, and S. Mancoridis. “A probabilistic approach to source code authorship identification”. // 4th International Conference on Information technology, IEEE Conference Publication, 2007.

[20] B. Alsulami, E. Dauber, R. Harang, S. Mancoridis, R. Greenstadt. “Source Code Authorship Attribution Using Long Short-Term Memory Based Networks”. // Proceedings of the 22nd European Symposium on Research in Computer Security, Oslo, Norway, 2017, pp. 65–82.

[21] A. Caliskan-Islam, R. Harang, A. Li, A. Narayanan, C. Voss, F. Yamaguchi, R. Greenstadt “De-anonymizing Programmers via Code Stylometry” // Proceedings of the 24th Usenix Security Symposium (2015)

[22] A System for Detecting Software Similarity http://theory.stanford.edu/~aiken/moss/ Retrieved: Dec, 2018

[23] JPlag Detecting Software Plagiarism https://jplag.ipd.kit.edu Retrieved: Dec, 2018

[24] The BOSS Online Submission System https://www.dcs.warwick.ac.uk/boss/ Retrieved: Dec, 2018