I.J. Intelligent Systems and Applications, **2017, 5, 27-33**
Published Online May 2017 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijisa.2017.05.04

Source Code Author Attribution Using Author's Programming Style and Code Smells

Muqaddas Gull
University of Sargodha, Sargodha, 40100, Pakistan
E-mail: Muqaddasgull@yahoo.com

Tehseen Zia and Muhammad Ilyas
COMSATS Institute of Information Technology, Islamabad, 44000, Pakistan, University of Sargodha, Sargodha, 40100, Pakistan,
E-mail: {tehseen.zia@comsats.edu.pk, m.ilyas@uos.edu.pk}

***Abstract*—**Source code is an intellectual property and using it without author's permission is a violation of property right. Source code authorship attribution is vital for dealing with software theft, copyright issues and piracies. Characterizing author's signature for identifying their footprints is the core task of authorship attribution. Different aspects of source code have been considered for characterizing signatures including author's coding style and programming structure, etc. The objective of this research is to explore another trait of authors' coding behavior for personifying their footprints. The main question that we want to address is that "can code smells are useful for characterizing authors' signatures? A machine learning based methodology is described not only to address the question but also for designing a system. Two different aspects of source code are considered for its representation into features: author's style and code smells. The author's style related feature representation is used as baseline. Results have shown that code smell can improves the authorship attribution.

***Index Terms*—**Authorship, Source Code, Stylistic Feature, Code Smell, Author style.

I. Introduction

Source code is an intellectual property and using it without author's permission is a violation of property right [1]. Source code authorship attribution is vital for dealing with software theft, copyright issues and piracies [2]. The broad area of study is known as software forensics [3]. A widely recognized way for performing attribution is through capturing and recognizing authors' signatures [4]. The domain of computer science that provides structures and algorithms for storing and identifying authors' footprints is known as machine learning. The effectiveness of this methodology is mainly relaying on defining signatures in terms of features. Different aspects of source code have been considered for characterizing signatures including author's coding style.

The objective of this research is to explore a new trait of authors' coding behavior for personifying footprints. Some authors are stricter to follow standard software engineering guidelines than others. This trait can be characterized by using code smells. The term "bad smell" is coined by Kent Beck to indicate that something somewhere in code has gone off beam. Each code smell corresponds to some standard software development guideline and presence of the smell indicates the lack of corresponding practice. The values of code smells indicate the degree of deficiency. The detail description of these code smells is given in [5]. While coding style can characterize regularities in authors' writing style, code smells can describe consistencies of authors' deviation from standard software engineering practices [6].

As coding style is a well-explored aspect for authors' signatures, it is considered as baseline method in this work. The main question that we want to address is that "can code smells are useful for characterizing authors' signatures? A machine learning based methodology is employed to address the question. The paper is organized as follows: related work is described in Section 2,the experimental methodology is presented in Section 3, results are presented in Section 4 and conclusion is described in Section 5.

II. Related Work

Authorship attribution of source code is different than attribution of text documents in different aspects; for instance, source code is not fully unstructured as text documents. The programming paradigm, language syntax, software engineering practices and development environments imposed the structure. Moreover, the software engineering practices force programmers to reuse the already developed code for common tasks. Furthermore, specific naming conventions for identifiers and functions used by development teams for consistency purposes brings back the impartiality of using authors' manifestation. Therefore, the standard techniques for attribution of text document (e.g. bag-of-words based features) may not be effective for source code. Majority of research on this topic is therefore focused on analyzing features for personifying authors.

In [2], nine software features from programming layout

& design are analyzed by using C4.5 classification algorithm. The features include i-as-iterator, Line Length, Comments, Average procedure length, Methods, Number of Arrays, Object Creation, Single literal variable and Double literal variable. In reported results the system has achieved an accuracy of 68.89%. In [5],the considered features are from three different categories including programming layout, programming structure and programming style. To analyze their effectiveness two classification algorithms are used including Gaussian classifier and MLP neural network. The achieved accuracy was 73% using MLP. They support the conclusion that for a limited set of programmer it is possible to identify a particular author and the probability to find that two programmers share exactly same characteristics should be very small. In another similar study [3], the author used stylist features including distribution of line size, leading spaces, underscores per line, semicolons, commas per line and tokens per line. The performance was analyzed by using naïve Bays and voting feature intervals. It is shown that VFI and naïve Bayes increase the success rate of classification.

In [6], the author used a variety of metrics categorizing them into three categories including programming style, programming layout and programming structure. A subset of effective features is obtained through a statistical analysis. Using a statistical approach known as SAS, the author has shown an overall classification accuracy of 73%. In [7],the effectiveness of operators, white space, literals, keywords, functions and I/O words has been shown by achieving an accuracy of 76.78% with these features. In [1], code stylometry based authorship attribution is performed by using JStylo tool. It is shown that SVM leads to highest accuracy and obtained the success rate varying from 70.31% to 90.91%.In [8], the author employed programming layout, style and structure metrics and compared the performance of three classification algorithms including discriminant analysis, neural network and case base reasoning. It is found that other statistical methods such as Bayesian technique can produce good result. In [4], the author evaluates the effect of discretization using a genetic algorithm. The author used four metrics including leading space, leading tab, line length and line word. The performance of GA is comparatively analyzed with other methods of discretization including range based discretization, frequency based discretization and no discretization. In reported results the system has achieved an accuracy of 60% for range based discretization, 70% for frequency based discretization and 65% with no discretization. In [9], the author used a variety of metrics including number of each type of data types, the cyclomatic complexity, quantity and quality of comments, type of variables and layout of code. They are also working on IDENTIFIED toolkit for automatic extraction of these metrics.

In [10], the author used a variety of metrics categorizing them into four categories including comments, programming layout features, identifiers and programming structure features. Comparative analysis is performed on two programming languages including java and common lisp and it is found that the selected metrics have strong influence for java programs but not for common lisp. In [11], the author propose to use lexical features such as variable names, formatting and comments, and some syntactic features such as presence of bugs and use of keywords for source code author attribution, but they don’t report any result or a case study experiment with a formal approach. In [12], the author presents a technique for automatic extraction of stylistic features from programs’ binaries. The effectiveness of N-grams, idioms, graphlets, supergraphlets, call graphlets and library calls has been shown with an accuracy of 77%. In [13], twenty six metrics are analyzed by using case-based reasoning, multiple discriminant analysis and feed forward neural network. The features include LOC, portion of letters that are uppercase, variables per line, portion of inline comments, pure comments, mean number of characters per line, cyclomatic complexity, if statements per NCLOC(Non comment per LOC), switch statements per NCLOC, while statements per NCLOC, decision statements per NCLOC etc. In reported results the system has achieved an accuracy of 88.0% in case of using case bases reasoning.

In [14], the author used a variety of metrics categorizing them into two categories including counting metrics e.g. leading spaces, trailing spaces, leading tabs, trailing tabs, line length, LOC, brace position, comments, average procedure length, average indentation, number of methods, unary and binary operators, number of loops, single literal variables, double literal variables and Boolean metrics e.g. i-as-iterator, conditional operator, try statements, naïve variable names and methods chaining. They have used decision tree algorithm for classification.

In this paper, we are presenting the analysis of an unexplored aspect of source code that deals with analyzing the code according to software development standards. The code smells are employed to model and quantify the code (and its author) according to the standard. Though no work is conducted for employing code smells for author attribution, code smells have been employed for other tasks such as software inspection [15] and improving software design [16].

III. Methodology

The authorship attribution of source code is posed as classification problem as shown in Fig. 1. Availability of a dataset is an essential requirement to develop and test the methodology. Many open source code repositories are available for accessing a set of source codes with designated authors e.g. planet-source-code [1], Source Forge[2], Codeplex[3] and GitHub[4], however most of these repositories contain source codes written by multiple authors. Since we want to analyze the efficacy of code smells for author attribution, we are interested in source

[1] http://www.planet-source-code.com/
[2] http://sourceforge.net/
[3] https://www.codeplex.com/
[4] https://github.com/

codes written by the single authors. One such readily available source for collecting the codeisplanet-source-code.com [17].The data set containing source code of nine different authors is collected from this source. The total gathered files are 153 including 42559 line of code (LOC).Distribution of LOC over authors is shown in Fig. 2. All the source code collected for our experiment was written in Java.

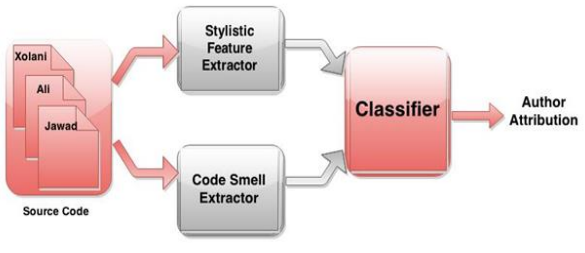


Fig.1. Block diagram of source code author attribution

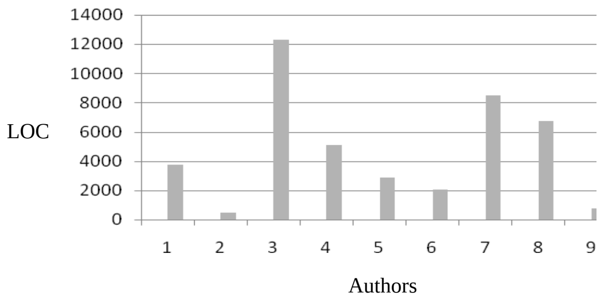
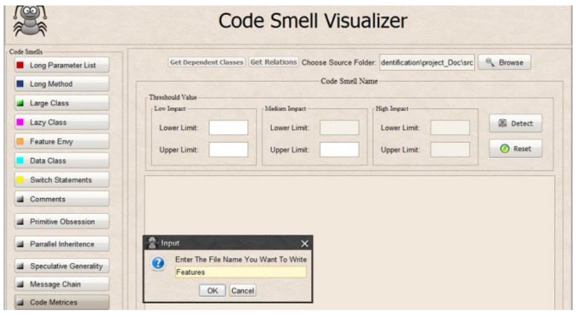


Fig.2. Distribution of LOC over authors

In the next phase, the code is represented in the form of features to generate a dataset. Two different aspects of source code are considered for the representation: author's coding style and code smells. The selection of style related features is based on related work [1-4, 7-14, 18-20]. However, the analysis of code smell related features is a novel contribution of this work. For feature extraction, two feature extractor modules are developed for each aspect of code as shown in Fig. 1. The style related feature extractor module (named as stylistic feature extractor) extracts features related with author's writing style. The code smell extractor module extracts features related to code smells. The interface of this module is shown in Fig. 3 and 4. Description of features is given in Section 3.1.To evaluate the performance of both feature representations, three well-known classifiers



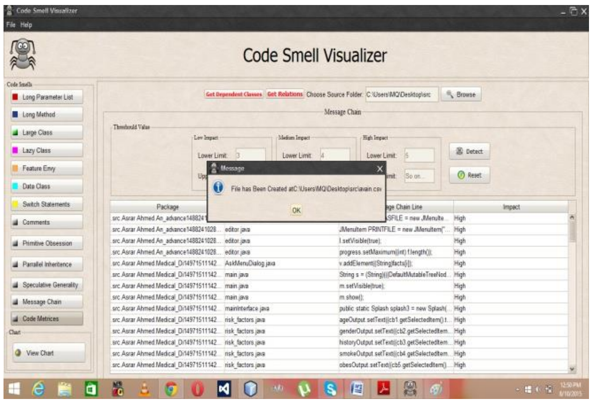
dentification\project_Doc\src

Enter The File Name You Want To Write

Features

Fig.3. A screenshot of stylistic feature extractor/visualizer module

are employed including naïve Bays, decision tree (i.e. J48), support vector machine and k nearest neighbors (KNN). A brief introduction of the classifiers is given in Section 3.2.



C:\Users\MQ\Desktop\src

Lower Limit: 3 | Lower Limit: 4 | Lower Limit: 5 | So on...

File has Been Created atC:\Users\MQ\Desktop\src\avain.csv

Package		...ge Chain Line	Impact
src.Asrar Ahmed.An_advance14882...		...SFILE = new JMenuIte...	High
src.Asrar Ahmed.An_advance1488241028...	editor.java	JMenuItem PRINTFILE = new JMenuItem("...	High
src.Asrar Ahmed.An_advance1488241028...	editor.java	l.setVisible(true);	High
src.Asrar Ahmed.An_advance1488241028...	editor.java	progress.setMaximum((int) f.length());	High
src.Asrar Ahmed.Medical_Di14971511142...	AskMenuDialog.java	v.addElement((String)facts[i]);	High
src.Asrar Ahmed.Medical_Di14971511142...	main.java	String s = (String)((DefaultMutableTreeNod...	High
src.Asrar Ahmed.Medical_Di14971511142...	main.java	m.setVisible(true);	High
src.Asrar Ahmed.Medical_Di14971511142...	main.java	m.show();	High
src.Asrar Ahmed.Medical_Di14971511142...	maininterface.java	public static Splash splash3 = new Splash(...	High
src.Asrar Ahmed.Medical_Di14971511142...	risk_factors.java	ageOutput.setText((cb1.getSelectedItem()).t...	High
src.Asrar Ahmed.Medical_Di14971511142...	risk_factors.java	genderOutput.setText((cb2.getSelectedItem...	High
src.Asrar Ahmed.Medical_Di14971511142...	risk_factors.java	historyOutput.setText((cb3.getSelectedItem...	High
src.Asrar Ahmed.Medical_Di14971511142...	risk_factors.java	smokeOutput.setText((cb4.getSelectedItem...	High
src.Asrar Ahmed.Medical_Di14971511142...	risk_factors.java	obesOutput.setText((cb5.getSelectedItem()...	High

12:50 PM 8/10/2015

Fig.4. A screenshot of code smell extractor/visualizer module

A. *Description of Features*

In this section, features are presented that are used for representing source code. Total number of considered features is 24 where 12 features are used to represent author's style and remaining 12 are used for representation of code smells. Description of style related features is given in Table 1.

Table 1. Features for representing author's style.

Sr. No	Name	Description
1	i-as-iterator	The feature counts occurrences of *i* as loop iterator variable in source code file. Some authors use variable *'i'* as an iterator in every loop while others intentionally avoid it. So, it represent the number of times the author used i as-iterator.
2	AvgLineLength(C)	It measures average line length in terms of characters of a source code file. Some authors have the habit of writing code into single line while some others write in multiple lines.
3	AvgLineLength(W)	It measures average line length in terms of words in source code file.
4	EL/Non-EL	It is the ratio of empty lines with non-empty lines. Empty lines are those lines which contain no character (not even a whitespace character) and non-EL contains one or more character other than newline character.
5	numTabs/length	This feature is a ratio between count of tab characters and total characters in a source code file and quantify the indentation of code using tab.
6	numSpaces/length	It is a ratio between count of space characters with total characters in a source code file and measures sparsityor density of code.

7	EL/length	The feature is the ratio of empty lines in a source code file with total characters. It is another measure to represent sparsity or density of code.
8	Uppercase	It is a percentage of uppercase/capital characters with total characters. One way to captures authors' variable name style.
9	Char/words	It is a ratio of total number of characters with total words in a source code file.
10	Spaces/SLOC-P	Spaces/SLOC-P represents ratio of total no of spaces count with SLOC-P in a source code file.
11	Statements/SLOC-L	It is a measure of ratio between total statements with SLOC-L in a source code file.
12	Words/statements	Words/statements represents ratio of total no of words count with total no. of statements in a source code file.

As code smells are the indications of potential problems in the software design, they are the symptoms of the problem which could be cured by refactoring. Different s*oftware metrics* are used to detect the code smells [5, 6]. We transformed the metrics into features to exploiting them for attribution of authors. Description of the features is given in Table 2.

Table 2. Features for representing code smells.

S.No	Name	Description
1	Long parameter list	When a piece of code has a method having a large number of parameters in its parameter list, it is said to have long parameter list [21]. This feature is a representation of this metric. As, some authors have the habit of using long parameter list while others not, so the feature can make a difference between such authors.
2	Long method	It is a measure of how much functionality functions are performing[21]. Usually it is determined on the basis of instance variables and number of lines of functions. As, some authors have the habit of writing short methods while others write long methods, so the feature can make a difference between such authors.
3	Large Class	It is a quantification of how much a class is loaded and often measures on the basis of instance variables, methods and LOC [21]. This feature can provide a differentiation about the authors that prefers to create small classes with those create large class.
4	Lazy Class	The term lazy class is used for referring the class that isn't doing much [22]. Lazy class is a class which is least fulfilling the purpose of creation. This feature is a quantification of laziness of classes and can differentiate between authors creating lazy classes or author that does not.
5	Feature Envy	When a method in a class is manipulating the data of another class rather than itself, the characteristic is known as feature envy code smell. The code smell is usually measured by identifying variables and methods of other classes accessed by that method. Existence of feature envy can make a difference between authors as either the author is using such methods or not.
6	Data Class	The term data class is used for class that only contains fields, getters and setters. Such a classis just a dump data holder and possesses no functionality. The code smell is usually measured as if a class only contains getter, setter methods, default constructor and data members; consider it as data class bad smell. Existence of this code smell can make a difference between authors i.e. some authors have the habit of creating such classes which only contain different getting and setting methods.
7	Switch Statements	When a number of switch statements are scattered throughout the source code it is indication of switch cases code smell [6]. The smell is quantified by counting all switch cases. If the switch cases count exceeds conditions and threshold values specified in our research work, consider this switch statement as switch statement bad smell. It can be an indication of an author specific behavior of using switch statement.
8	Comments	Burdening the code with comments is also considered as a code smell [21].The smell is measured by counting comment lines if it contains comments greater than 18%, consider it as comment bad smell. Some authors have the habit of writing excessive comments. Such authors can be differentiated based on the smell.
9	Primitive Obsession	Primitive Obsession is using primitive data type to represent domain ideas [6] as we use String to represent date instead of using Date class. The code smell is usually measured by obtaining the number of primitive variables of each class and calculating their average. All the classes having NOV (Number of variables) greater than the counted average consider that primitive obsession code smell exists. Existence of this code smell can make a difference between authors as either authors prefer to use primitive data types or to use user defined data types.
10	Parallel Inheritance	In the case of this smell, every time a subclass of a class is made, one also has to make a subclass of another [6]. It is measured by obtaining DIT(Depth of Inheritance) and NOC (number of children) at each level for all the classes. Find DIT having depth greater than two and classes having similar DIT and NOC values at corresponding levels mean the code smell exist. Existence of this code smell can make a difference between authors either author is creating such subclasses of a class or not.
11	Speculative Generality	The code smell describes the existence of unnecessary block of code [21]. For example, sometimes classes and methods are simply created for future enhancement. Existence of the smell is measured by capturing the abstract classes that have not been implemented anywhere in the system. We can capture the author behavior because of this code smell either author is creating abstract classes for future enhancement or the author prefer to create such classes when required.

12	Message Chain	A situation when there is a chain of messages passing from one object to another gives birth to "message chain" code smell [6]. It is quantified as capture the set of classes and objects if we need to call three objects to get a method so it is cindered as message chain code smell. The author behaviors can be captured as either they are using multiple objects call to call a method or they prefer to call objects closely related to it.

B. Classifier

Third phase of methodology is to train classifiers for categorizing authors. Four well-known classifiers are used including naïve Bayes, decision tree (J48), support vector machine (SVM) and k nearest neighbors (kNN).

SVM is a popular classification algorithm and known for its robustness and efficiency [24]. The major causes of its success include sound theoretical basis, reliance only on few training instances and insensitivity against outliers. While learning to classify source codes into authors, each code can be represented as a vector in feature space. Then decision hyper-planes can be learned between different authors with the support of closest instances/vectors (known as support vectors)of authors. SVM also warranties to deliver maximum margin hyper-plane. The term margin is defined as a distance of supporting vectors from hyper-plane.

In naive Bayes classifier, Bayesian theorem is employed to find posterior probability of authors given source code [25]. The computational efficiency of classifier is relied on an assumption known as class conditionally independence which states that features are independent with each other, given the class. Though the assumption may not be realistic in most of settings (e.g. words in text) yet the classifier has shown its effectiveness in various applications.

The decision tree (DT) classifier learns a hypothesis in the form of tree. Each node of the tree relates to a feature and each edge of the node represents a test on feature value. Leafs of the tree correspond to classes (i.e. authors). Once the DT is learned, the classification decision is made by passed source code through different tests on the features starting from the root node until the leaf node is reached where label of the node (i.e. an author) is assigned to the document [25].

Unlike previously described algorithms, k nearest neighbor (KNN) algorithm don't learn an explicit hypothesis rather the training dataset is kept in memory. The classification decision is made on the basis of majority voting of nearest neighbors of given instance. Since there is no generalization beyond the training instances in such methods, they are known as lazy learners. As the dataset is revisited each time a classification decision is performed, the method is computationally expensive. Despite that, the method is widely studied and quite effective for text categorization [26]. This is the reason; we have chosen it to study its performance with other classifiers.

IV. Experimentation and Results

To gauge the performance of a classifier, a standard f-measure is used as defined below.

$$F\ measure = \frac{2pr}{p+r} \qquad (1)$$

Where p is precision and r is recall. Moreover, k-fold cross-validation test is applied to validate the results. Experimentations are performed in WEKA (an open source machine learning toolkit) [27]. For SVM classifier, we used a wrapper of standard LIBSVM 3.17 [28] toolkit for WEKA.

To access the efficacy of author's style and code smells, three test cases are analyzed. Results of all the test cases are collectively shown in Fig. 5. First test case is to access proficiency of author's style related features. Second test case is to measure the effectiveness of code smells. Third test case is to evaluate the performance of style related features and code smells collectively. Since each classifier makes some assumptions about dataset, comparatively analysis of classifiers is also an objective. It can be seen from results that the performance of classifiers is increased when both stylistic features and code smells are collectively used. Moreover, naïve Bays classifier has shown an advantage though conditional independence assumption may seems unrealistic in this setting.

It may be because of two reasons: firstly, naive Bays converge quicker than discriminative models and secondly as we have a moderate size dataset, being high bias/low variance classifiers, naive Bays has an advantage over low bias/high variance classifiers (such as kNN) [29]. These results lead us to conclude that code smells can provide an additive advantage when considered as complementary with stylistic features. These results show that in these approaches, as only using stylistic features, only using code smells and using both stylistic features and code smell the naïve Bayes classifier performs really well, having the highest accuracy when it is compared with J48 , support vector machines and KNN.

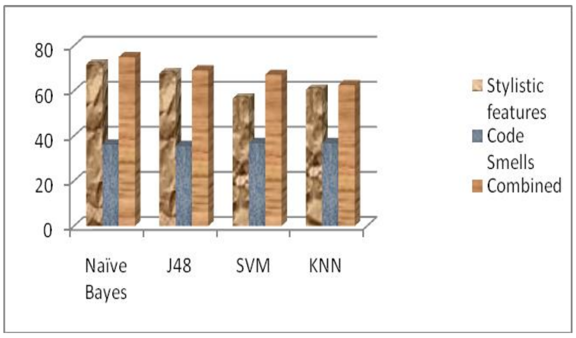


Fig.5. Performance of style based, code smells and combined feature representations of source code attribution across classifiers.

V. Conclusion

The problem of authorship attribution using stylistic features and code smells is considered. Author attribution is to identify the author of a code and this can be done by the process of feature extraction. Two different aspects of source code are considered for its representation into feature, one is stylistic features which are regarding the author style and the other is code smells which is the novel contribution of this work to use code smells along with stylistic features. To access the efficacy of author's style and code smells, two test cases are analyzed: First test case is to access proficiency of author's style related features. Second test case is to evaluate the performance of style related features and code smells collectively. It has been shown that the extracted features were contributive for authorship attribution, with the classification accuracy of 75% by augmenting code smells along with stylistic features. Comparative analysis of well-known classifiers: Naive bayes, SVM, J48 and KNN is also performed and the result shows that in both test cases, Naïve Bayes classifier performs really well, having the highest accuracy when it is compared with other classifiers. In future, multi-author attribution of source code will be performed based on the analysis of this paper.

References

[1] A. C. Islam, " Poster: source code authorship attribution," Comput. Cardiol IEEE Press, 1997.

[2] R. R. Joshi, R. V. Argiddi, "Author identification: an approach based on style feature metrics of software source codes," International Journal of Computer Science and Information Technologies, vol. 4, no. 4, 2013.

[3] A. Gray, P. Sallis and S. MacDonell, "Software forensics: extending authorship analysis techniques to computer programs", in Proceedings of the 3rd Biannual Conference of the International Association of Forensic Linguists (IAFL), 1997.

[4] J. Kothari, M. Shevertalov, E. Stehle, and S. Mancoridis, "A probabilistic approach to source code authorship identification," In 4th International Conference on Information technology, IEEE, 2007, pp. 243–248.

[5] M. Mantyla, J. Vanhanen and C. Lassenius, "A taxonomy and initial empirical study of bad smells in code", in Proceedings of the IEEE International Conference on Software Maintenance, pp. 381-384, 2003.

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts," Refactoring: improving the design of existing code," New Jersey: Addison-Wesley, 2000.

[7] S. Burrows, A. L. Uitdenbogerd, and A. Turpin, "Application of information retrieval techniques for source code authorship attribution," Fourteenth International Conference on Database Systems for Advanced Applications, April 2009, pp. 699-713.

[8] G. Frantzeskou, S. Gritzalis and S. G. MacDonell, "Source code authorship analysis for supporting the cybercrime investigation process," in 1st International Conference on E-Business and Telecommunication networks, 2004, pp. 85-92.

[9] M. Shevertalov, J. Kothari, E. Stehle, and S. Mancoridis, "On the use of discretized source code metrics for author identification," in 1st International Symposium on Search Based Software Engineering, 2009, pp. 69-78.

[10] G. Frantzeskou, S. MacDonell ,E. Stamatatos and S. Gritzalis, "Examining the significance of high-level programming features in source code author classification," in Journal of System and Software, vol. 81,no. 3, pp. 447-460, 2008

[11] E. H. Spafford and S. A. Weeber, "Software forensics: can we track code to its authors?," Computers & Security, vol. 12, no. 6, 1993 pp. 585-595.

[12] N. Rosenblum, X. Zhu, and B. P. Miller, "Who wrote this code? Identifying the authors of program binaries," Computer Security–ESORICS 2011, 2011, pp. 172–189.

[13] N. Rosenblum, X. Zhu, and B. P. Miller, "Software forensics applied to the task of Discriminating between Program Authors," in Journal of System Research and Information Systems 10, 2001, pp. 113-127.

[14] R. R. Joshi, R. V. Argiddi and S. Sulabha, "Author identification: an approach based on code feature metrics using decision trees," in International Journal of Computer Applications (0975-8887), vol. 66, no.4, March 2013.

[15] F. A. Fontana, P. Braione and M. Zanoni, "Automatic detection of bad smells in code: An experimental assess," in Journal of Object Technology, vol.11, no.2, 2012.

[16] I. Krsul and E. H. Spafford, "Authorship analysis: identifying the author of a program," in proceeding of the 8th national Information System Security Conference, National Institute of Standard and Technology, 1995, pp. 514-524.

[17] [Online]. Available: https://sourcemaking.com/refactoring/lazy-class

[18] I. Krsul and E. H. Spafford, *Authorship analysis: identifying the author of a program*, Technical Report TR-96-052, September 1996.

[19] A. Gray, and S. MacDonell, "Identified: A dictionary-based system for extracting source code metrics for software forensics," in Proceeding of Third Software Engineering: Education and Practice International Conference, IEEE, pp. 252-259, 1998.

[20] M. A., Cusumano & R. W. Shelby, "Microsoft secrets," New York: NY, 1995.

[21] M. Fowler, K. Beck, J. Brant, W. Opdyke,D. Roberts. Refactoring Improving the Design of Existing Code, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA 1999 ISBN:0-201-48567-2

[22] T. W. Kim, T. G. Kim and J. H. Seu, "Specification and automated detection of code smells using OCL," in International Journal of Software Engineering and Its Applications, vol. 7, no. 4, July 2013.

[23] A. Chatzigeorgiou and A. Manakos," Investigating the evolution of bad smells in object-oriented code," in International conference on the Quality of Information and Communications Technology (QUATIC), IEEE, pp. 106–115, 2010.

[24] Y. Lin, "Support vector machines and the bayes rule in classification, Data Mining and Knowledge Discovery, vol. 6, no. 3, pp. 259–275, 2002.

[25] G. Dimitoglou, J. A. Adams and C. M. Jim, "Comparison of the C4.5 and a Naïve Bayes Classifier for the Prediction of Lung Cancer Survivability," Journal of Computing ,vol. 4, Issue 8, August 2012

[26] M.L Zhang, "A k-nearest neighbor based algorithm for multi-label classification" IEEE International Conference on Granular Computing, 2005, pp: 718-721.

[27] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I. H. Witten, "The WEKA data mining software: an update", ACM SIGKDD Explorations Newslette, vol. 11, pp: 10-18, 2009.

[28] C. C. Chang and C. J. Lin, "LIBSVM: A Library for support vector machines," in ACM Transactions on Intelligent Systems and Technology ,vol. 2, no. 3, 2011.
[29] H. Zhang, "The Optimality of Naive Bayes", American association for artificial intelligence, 2004.

Authors' Profiles

Tehseen Zia received the Ph.D degrees in Computer Science from Vienna University of Technology, Austria. He is currently working as Assistant Professor in COMSATS Institute of Information Technology, Islamabad. His research interests include machine learning and data mining.

Muhammad Ilyas received the Ph.D in Software Engineering from Johannes Kepler University Linz. He is currently working as Assistant Professor in Department of Computer Science, University of Sargodha. His research interests include Software Engineering, Human Computer Interaction and Artificial Intelligence.

Muqaddas Gull received her B.S degree in Software Engineering from University of Sargodha in 2013. She has received her M.S degree in Computer Science from University of Sargodha in 2015.
Her research interests include machine learning, data mining and software requirement engineering.

How to cite this paper: Muqaddas Gull, Tehseen Zia, Muhammad Ilyas,"Source Code Author Attribution Using Author's Programming Style and Code Smells", International Journal of Intelligent Systems and Applications(IJISA), Vol.9, No.5, pp.27-33, 2017. DOI: 10.5815/ijisa.2017.05.04