Full citation: MacDonell, S.G., & Gray, A.R. (2001) Software forensics applied to the task of discriminating between program authors, Journal of Systems Research and Information Systems 10, pp.113-127.

Software Forensics applied to the task of Discriminating between Program Authors

Stephen G. MacDonell and Andrew R. Gray

Software Metrics Research Lab
Department of Information Science
University of Otago, PO Box 56, Dunedin, New Zealand
+64 3 4798135 (ph.) +64 3 4798311 (fax), stevemac@infoscience.otago.ac.nz

Abstract

Software forensics is here regarded as the particular field of inquiry that, by treating pieces of program source code as linguistically and stylistically analyzable entities, attempts to investigate various aspects of computer program authorship. These inquiries could be performed with any number of goals in mind, including those of identification, discrimination, and characterization of authors. In this paper we extract a set of 26 authorship-related metrics from 351 source code programs, written by 7 different authors. The use of feed-forward neural network (FFNN), multiple discriminant analysis (MDA), and case-based reasoning (CBR) models for discriminating these programs are then investigated in terms of classification accuracy for the authors on both training and testing (holdout) samples. The first two techniques (FFNN and MDA) produce remarkably similar results, with the overall best results coming from the CBR models. All of the examined modeling techniques have prediction accuracy rates over 80%, supporting the claim that it is feasible to use such techniques for the task of discriminating program authors based on source-code measurements in a majority of cases.

1. INTRODUCTION

In a number of quite varied situations there is a desire, or even a requirement, to be able to investigate the nature of a computer program's authorship. By this it is meant, that there is some question concerning the authorship of a series of programs or alternatively the characteristics of program authors [3]. While computer program source code is certainly much more formal and restrictive than spoken or written languages in terms of its grammar and expressiveness, computer programmers are still able to make use of quite large degrees of flexibility when writing a program to achieve a particular purpose. This flexibility includes such factors as:

- the manner in which the task is achieved (the algorithm used to solve the problem),
- the way that the source code is presented in terms of layout (spacing, indentation, bordering characters used to set off sections of code, standard headings, etc.), and
- the stylistic manner in which the algorithm is implemented (the particular choice of program statements used where there is a choice, variable names, etc.).

Other options that may also be available to the programmer include selecting the computer platform, programming language, compiler, and/or text editor to be used. These decisions may also allow the programmer some degree of personal freedom, and thus expressiveness. Many of the features of a computer program (algorithm, layout, style, and environment) can be quite specific to certain programmers or types of programmer. Ideally, such aspects in order to be useful for software authorship analysis have low within-programmer variability, and high between-programmer variability. This is especially likely for particular combinations of features and unusual programming idioms that generally make up a programmer's problem-solving vocabulary. Therefore, it seems quite plausible to suggest that computer programs can contain some degree of information that provides evidence of the author's identity and characteristics.

The most widely known example of authorship analysis is plagiarism detection, usually in an academic setting, where students' assignments can be compared to see if some are "suspiciously similar" [7]. The incidence of highly similar programs can provide suggestive evidence that one student's code may have been derived from another student's work. This particular area of research provided the origins of the ideas that now make up the field of software forensics—which is defined here as the study of program characteristics with the intention of identifying, examining, or discriminating between program authors [1].

Software forensics also includes the areas of authorship characterization, as in psychological studies of the relationships between programmer attributes and their code and between programming conditions and code. The analysis of malicious code (such as computer viruses and Trojan horses) is another application area, although this generally involves much more subjective analysis [6].

Other less common applications of software forensics include quality control (through coding standards for example, cyclomatic complexity or comment density), author tracking (for example, determining the author of code of unknown origin), change control (tracking the authorship of changes and quality control when making changes), and ownership disputes.

While the idea of dissenting the structure and nature of programs to discern some information about the likely author or authors and/or their characteristics may appear somewhat esoteric, perhaps even unrealistic, it has been shown that such activities are feasible, at least under certain circumstances [2]. In fact many measurements may even be difficult for programmers to change [6]. An open question is how such models should be constructed to best represent the mappings between program features, authors, and the authors' characteristics.

In this paper the focus will be on the area of developing models that are capable of discriminating between several authors using source-code based measurements. The measurements that are preferred here are those that can be automatically extracted from source code by pattern matching algorithms since the volumes of data needed for these applications will generally surpass convenient human measurement. In some cases however, expert intervention may be necessary where measures cannot be usefully extracted in an automated way, for example using fuzzy logic categories to express the correspondence between comments and code behavior.

2. APPLICATIONS OF SOFTWARE FORENSICS

In this section some additional information about each of the four main applications of software forensics are provided, namely: author identification, author discrimination, author characterization, and author intent determination. The models developed later in the paper are concerned only with the first of these four applications, but it is useful to keep the others in mind as the same techniques can also be applied to these problems.

2.1 Author identification

The goal here is to determine the likelihood of a particular author having written some piece(s) of code, usually based on other code samples from that programmer. This can also involve having samples of code for several programmers and determining the likelihood of a new piece of code having been written by each programmer. This application area is very similar to, for example, the attempts to determine the authorship of the Shakespearean plays or certain biblical passages. An example of this applied to source code would be ascribing authorship of a new piece of code, such as a computer virus, to an author where the code matches the profile of other pieces of code written by this author.

2.2 Authorship discrimination

This is the task of deciding whether some pieces of code were written by a single author or by (some number of) different authors. This can possibly also include an estimate of the number of distinct authors involved in writing a single piece or all pieces of code. It is obviously necessary to distinguish between identifying multiple authors for a series of programs and co-authorship on a single program. This task involves the calculation of similarity between the two or more pieces of code and possibly some estimate of between-and within-subject variability. An example of this would be showing that different authors, without actually identifying the authors in question, probably wrote the two or more pieces of code.

2.3 Author characterization

This is based on determining some characteristics of the programmer of a code fragment, such as personality and educational background, based on their programming style. An example of this would be determining that a piece of code was most likely to have been written by someone with a particular educational background due to the programming style and techniques used.

2.4 Author intent determination

It may be possible to determine, in some cases, whether code that has had an undesired effect was written with deliberate malice, or was the result of an accidental error. Since the software development process is never error free and some errors can have catastrophic consequences, such questions can arise reasonably frequently. This can also be extended to check for negligence, where erroneous code is perhaps suspected to be much less rigorous than a programmer's usual code. This is a much-neglected aspect of source code authorship analysis with no other literature found that mentions its use. While this could be seen as the most difficult, and certainly the most subjective, of the applications it may also be one of the most crucial in practice.

3. SETTINGS FOR USING SOFTWARE FORENSICS

In order to further provide a context for the application of the modeling techniques to authorship identification, here a number of application areas are discussed as potential situations motivating the need for such an exercise, as well as possible applications of the other software forensics problems.

3.1 Educational

The educational setting of software forensics is generally concerned with plagiarism detection. A significant amount of literature has been produced detailing various schemes for detecting cases where programming assignments have been plagiarised, with or without the original author’s consent. Generally plagiarism detection is a combination of author identification (who really wrote the code), and author discrimination (did the same person write both pieces of code). One significant problem that emerges when using plagiarism detection is the effect of discouraging collaboration between students. Other issues such as student’s adopting tutors, lecturers, and/or textbook author’s styles are also problematic.

3.2 Legal

The use of software forensics for tracking down the authors of malicious code has been the second most emphasized application after plagiarism detection. Other issues such as the intent analysis of malicious code also appear under this heading.

3.3 Industrial

Within an industrial context there are fewer applications of software forensics, but cases would include identifying authors of code that needs to be maintained where this information is not otherwise recorded or may be incorrect, and checking for negligent programming.

3.4 Psychological

While the above areas are mostly practical, there are also several uses for software authorship analysis from a theoretical perspective. It is possible to use such metrics to examine the developmental process of programming skills, and to correlate individual characteristics to programming ones.

4. MEASUREMENTS FOR SOFTWARE FORENSICS

Expert opinion can, potentially, be given on the degrees of similarity and difference between code fragments, although this is likely to be a time-consuming exercise in many cases. Psychological analysis of code can also be performed, even as a simple matter of opinion. However, a more scientific approach may also be taken (and should be taken) since both quantitative and qualitative measurements can be made on computer program source code and object code. These measurements can be either automatically extracted by analysis tools (such as IDENTIFIED, discussed later in the paper), calculated by an expert, or arrived at by using some combination of these two methods. Some metrics can obviously only be calculated by an expert, such as the degree to which the comments in code match the actual behavior of that code. Here these measurements are referred to as metrics for reasons of tradition and include some borrowed and adapted from conventional software metrics and linguistics. A vast number of different metrics can be extracted from source code, although some are obviously more likely to be effective than others.

5. TECHNIQUES FOR AUTHORSHIP DISCRIMINATION

5.1 Neural Networks

There are a vast number of neural network architectures and training algorithms contained within the literature. The most commonly used architecture for applications is that of a feed-forward neural network (FFNN), which is still generally trained using some modified form of the gradient-descent algorithm.

The main issues when using this approach concern selecting the optimal architecture for the network and in stopping the training (usually by using data set splitting and stopping training when a validation data set error is minimized). The use of data set splitting can be seen as a disadvantage, since this reduces the amount of data available for the network to learn the relationships.

More sophisticated approached that do not require hold-out samples are not investigated here as they are likely to be less accessible to researchers in applied fields.

5.2 Discriminant Analysis

Multiple discriminant analysis (MDA) is a statistical technique that separates observations into two or more groups based on several orthogonal linear functions of the independent variables. The technique assumes a reasonable degree of multivariate normality, with logistic regression an alternative where this is not the case.

A significant advantage of discriminant analysis as a technique is the easy availability of stepwise procedures for controlling the entry and removal of variables. By working with only those necessary variables we increase the chance of the model being able to generalize to new sets of data. In addition, the data collection costs can be reduced, sometimes significantly, by working with a smaller set of variables.

Another advantage of the technique is that it provides probability information for the predictions, both in terms of the conditional probability of an observation belonging to a particular class given its classification and the conditional probability that a particular observation will be classified as belonging to a particular class given its real class. In a legal setting such information would certainly be required if software forensic results were to be accepted as evidence.

5.3 Case-Based Reasoning

Case-based reasoning (CBR) is a method for modeling the relationship between a series of independent variables and one or more dependent variables by storing the cases (observations) in a database. When presented with a new observation, the cases that are similar in terms of the independent variables are retrieved and the dependent variables calculated from them using some form of “averaging” process.

CBR has the advantages of not requiring any distributional assumptions *per se* but does require the specification of a distance metric (for finding the closest exemplars to the presented case and calculating their similarity). Scaling (if any is used) when measuring similarity can be based on ranges or standardized values if some distributional assumptions are made.

The other aspect that requires some thought is the selection of a method for combining the cases. Again, a simple weighted average approach can be used once the distance metric has been decided on, with perhaps some power of distance used to increase the influence of closer observations and reduce the influence of outliers. In most implementations a threshold of similarity or a limit of "related" cases is used to prevent all stored cases influencing all predictions.

One particular case-based reasoning system that has been previously used for software metric research is the ANGEL system [5]. ANGEL has also been implemented as part of the IDENTIFIED system that was used in this paper for the measurement extraction, and CBR and FFNN models [1, 4]. The ANGEL system also allows for the automatic selection of relevant variables (at some considerable computational cost), although here no attempt will be made to select any optimal subset of variables when using this technique.

6. AUTHORSHIP DATA SET

The data that we have chosen to illustrate the author discrimination problem exhibits many of the characteristics that present some of the most perplexing difficulties found when undertaking such analyses. These difficulties include small amounts of data, unequal amounts of data from different authors, and code from some authors varying over time and application domain.

The data set used here contains programs from seven authors with widely varying amounts of data and from three basic source types. 26 measures were extracted for each program using the IDENTIFIED tool (Table 1).

All programs were written in standard C++. The source code for authors one, two, and three are from programming books; authors four, five, and six are experienced commercial programmers; and author seven's code is from examples provided with a popular C++ compiler. The choice of program sources may appear unusual, but it was felt that the usual source of student programs was no more realistic.

For the purposes of testing the various models to be developed in sections 7.1, 7.2, and7.3, the available data was split (as shown in Table 2) with stratification (as equally as possible) across authors. The split was approximately 25% in the Training 1 set, 25% in the Training 2 set, and 50% in the Testing set.

In some cases, especially for authors 4 and 5, very little data is available, but this can be seen as a useful test of a situation certain to arise in practice. The only concern here is that the prior probabilities from the Training set match the posterior probabilities in the Testing set.

In a simulation-based study the use of resampling would appear a better choice to assess the techniques. However since this study involves only one split of the data set, the use of stratification seems preferable to the increased effects of chance bought on by resampling.

Measurement	Description
WHITE	Proportion of lines that are blank
SPACE-1	Proportion of operators with whitespace on both sides
SPACE-2	Proportion of operators with whitespace on left side
SPACE-3	Proportion of operators with whitespace on right side
SPACE-4	Proportion of operators with whitespace on neither side
LOCCHARS	Mean number of characters per line
CAPS	Proportion of letters that are upper case
LOC	Non-whitespace lines of code
DBUGSYM	Debug variables per line of code (LOC)
DBUGPRN	Commented out debug print statements per LOC
COM	Proportion of LOC that are purely comment
INLCOM	Proportion of LOC that have inline comments
ENDCOM	Proportion of end-of-block braces labeled with comments
GOTO	Gotos per non-comment LOC (NCLOC)
COND-1	Number of #if per NCLOC
COND-2	Number of #elif per NCLOC
COND-3	Number of #ifdef per NCLOC
COND-4	Number of #ifndef per NCLOC
COND-5	Number of #else per NCLOC
COND-6	Number of #endif per NCLOC
COND	Conditional compilation keywords per NCLOC
CCN	McCabe's cyclomatic complexity number
DEC-IF	if statements per NCLOC
DEC-SWITCH	switch statements per NCLOC
DEC-WHILE	while statements per NCLOC
DEC	Decision statements per NCLOC

Table 1: The 26 variables used

	Author							
Data set	1	2	3	4	5	6	7	Total
Training 1	17	29	7	3	1	11	21	89
Training 2/Validation	17	28	6	3	2	10	21	87
Testing	34	57	13	6	2	21	42	175
Total	68	114	26	12	5	42	84	351

Table 2: Data set splits

7. RESULTS

The results for each of the three modeling techniques are now discussed in turn. In each case confusion matrices are provided to emphasize that different techniques may give the same or similar overall performance in very different ways.

7.1 Neural Network

The ultimately selected FFNN was a 26-9-7 network, with the logistic transfer for both hidden and output layers. The best network found was trained for 250 epochs using the back-propagation algorithm (learning rate 0.2, momentum 0.9). All 26 variables provided were used. Half of the training data (Training 1) was used for the actual training, while the remainder (Training 2) was used to stop training and select the best architecture.

		Predicted author number							
		1	2	3	4	5	6	7	Total
Actual author number	1	$\boxed{20}$	1	6	1		1	5	34
	2		$\boxed{57}$						57
	3			$\boxed{13}$					13
	4		2		$\boxed{4}$				6
	5		2			$\boxed{0}$			2
	6	1	2	1			$\boxed{17}$		21
	7	4	3	4				$\boxed{31}$	42
	Total	25	67	24	5	0	18	36	175

Table 3: Confusion matrix for testing data predictions from FFNN model using all training data

Table 3 shows the confusion matrix for the network's predictions on the testing set. Those programs that were correctly classified are shown as boxed entries on the main diagonal. As can be seen the network has a high classification rate of 81.1%. Authors two and three are obviously distinct from all others, while the small amount of data available for author five seems likely to be responsible for all of those programs being misclassified.

Since this technique was the only one that required splitting the training data, all other techniques were developed using both training data sets (Training 1 and 2) and just the first 50% (Training 1). The other modeling techniques when tuned using both training data sets could be expected to enjoy an advantage over the neural network model in terms of the greater number, and thus richness, of cases available. While in the second case the neural network models should have an advantage since they are tuned on the same data set whilst having their generalizability encouraged by the use of the validation set. Section 7.4 shows the performance of all models on all (sub)sets of data.

7.2 Multiple Discriminant Analysis

The MDA was a stepwise MDA (Wilk's lambda was used for entry and exit of variables). Prior probabilities were obtained from the data and within group covariance matrices were used. As discussed in section 7.1 both sets of training data were used as part of the model parameter tuning since no model selection process was used. Another model was developed using only the Training 1 data set (50% of the training data). See section 7.4 for these results.

Table 4 shows the confusion matrix for the predictions made on the with-held testing data. As with the neural network model the performance accuracy is 81.1% when using all training data. The patterns of confusion are similar for authors four, six, and seven but rather different for the other authors.

		Predicted author number							
		1	2	3	4	5	6	7	Total
Actual author number	1	$\boxed{26}$	1			3	1	3	34
	2	2	$\boxed{52}$		1		2		57
	3	1	2	$\boxed{10}$					13
	4		2		$\boxed{4}$				6
	5				1	$\boxed{0}$		1	2
	6	2	2	1			$\boxed{16}$		21
	7	3	3	2				$\boxed{34}$	42
	Total	34	62	13	6	3	19	38	175

Table 4: Confusion matrix for testing data predictions from MDA model using all training data

		Predicted author number							
		1	2	3	4	5	6	7	Total
Actual author number	1	$\boxed{28}$	1	2				3	34
	2		$\boxed{57}$						57
	3			$\boxed{13}$					13
	4		2		$\boxed{4}$				6
	5		1			$\boxed{1}$			2
	6		5				$\boxed{16}$		21
	7	1	5	2	2			$\boxed{32}$	42
	Total	29	71	17	6	1	16	35	175

Table 5: Confusion matrix for testing data predictions from CBR model using all training data

7.3 Case-Based Reasoning

The case-based reasoning model was developed using the ANGEL algorithm, with 5 analogies and weighted means for case aggregation. Tie resolution was also used. All variables were normalized in order to maintain a comparable scale.

All 26 variables were used, with two models developed – one using only 50% of the training data (Training 1) and another using all training data (Training 1 and 2). See section 7.4 for a discussion of the performance of this reduced-data model.

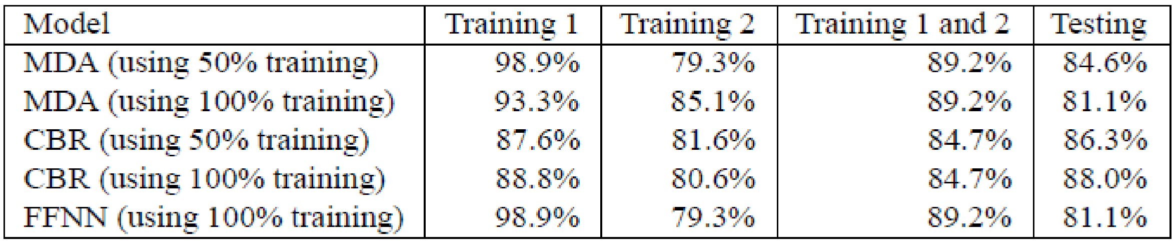
Table 5 shows the confusion matrix for the testing data set. There is a considerably higher level of accuracy compared to the neural network and discriminant analysis models, with 88.0% accuracy achieved when using all training data.

7.4 Comparison

Table 6 shows the results for all five models developed. Note that the “training set” errors for the CBR models are leave-one-out since the case to be predicted should obviously not be in the training set. As can be seen the results for the FFNN and MDA models are quite remarkably almost identical (the FFNN and full-data MDA are in fact identical). However, each of these models made rather different patterns of confusion on all data sets.

The best performing technique in all cases is case-based reasoning. In terms of predictive performance on the test data set, its predictions were almost 7% better which appears to be a useful increase in performance. Even with the reduced training data set, the case-based reasoning model outperformed the neural network model by 5.2%.

This is suspected to be a result of the fact that programmers have more than one style of programming leading to several multi-dimensional “clouds” of points. Some sets of programs for a given programmer are apparently within other programmer’s “clouds” of metrics, preventing simple explicit classification boundaries from properly classifying the systems.



Model	Training 1	Training 2	Training 1 and 2	Testing
MDA (using 50% training)	98.9%	79.3%	89.2%	84.6%
MDA (using 100% training)	93.3%	85.1%	89.2%	81.1%
CBR (using 50% training)	87.6%	81.6%	84.7%	86.3%
CBR (using 100% training)	88.8%	80.6%	84.7%	88.0%
FFNN (using 100% training)	98.9%	79.3%	89.2%	81.1%

Table 6: Results for discriminating models

8. CONCLUSIONS

The use of the proposed set of metrics for discriminating between seven authors shows promising results, especially when using the case-based reasoning technique. All techniques however provided accuracy between 81.1% and 88.0% on a holdout testing set would be certainly encouraging for the software forensics field as a whole.

It is tentatively suggested here that the nature of class boundaries for forensic applications is more amenable to modeling using case-based reasoning than partitioning approaches. The idea of multiple clusters suggests that other neural network architectures such as variants of LVQ could be fruitfully applied here.

We are now comparing the performance of different sets of forensic metrics, both structural and stylistic to determine which are the most useful in certain circumstances. Since stylistic metrics are easier to fake than structural, the ability of the latter to discriminate authorship is more useful.

Another area of interest is how each technique performs given certain quantities of data. Whilst the CBR models were better here it would seem likely that their performance would suffer more from losing data when compared to models using actual classification boundaries.

ACKNOWLEDGMENTS

The authors would like to thank Grant MacLennan and Philip J. Sallis for their help with an earlier version of the paper.

REFERENCES

[1] A. Gray, P. Sallis, and S. MacDonell. Identified (integrated dictionary-based extraction of non-language-dependent token information for forensic identification, examination, and discrimination): A dictionary-based system for extracting source code metrics for software forensics. In *Proceedings of SE:E&P’98 (Software Engineering: Education and Practice Conference)*, pages 252–259. IEEE Computer Society Press, 1998.

[2] I. Krsul and E. H. Spafford. Authorship analysis: Identifying the author of a program. Computers & Security, 16(3):233–256, 1997.

[3] P. Sallis, A. Aakjaer, and S. MacDonell. Software forensics: Old methods for a new science. In *Proceedings of SE:E&P’96 (Software Engineering: Education and Practice)*, pages 367–371. IEEE Computer Society Press, 1996.

[4] P. Sallis, S. MacDonell, G. MacLennan, A. Gray, and R. Kilgour. Identified: Software authorship analysis with case-based reasoning. In *Proceedings of the Addendum Session of the 1997 International Conference on Neural Information Processing and Intelligent Information Systems*, pages 53–56, 1998.

[5] M. Shepperd and C. Schofield. Estimating software project effort using analogies. *IEEE Transactions on Software Engineering*, 23(11):736–743, 1997.

[6] E. H. Spafford and S. A. Weeber. Software forensics: Can we track code to its authors? *Computers & Security*, 12:585–595, 1993.

[7] G. Whale. Software metrics and plagiarism detection. *Journal of Systems and Software*, 13:131–138, 1990.