ARL-TN-0860 ● DEC 2017

US Army Research Laboratory

Source-Code Stylometry Improvements in Python

by Gregory Shearer and Frederica Nelson

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TN-0860 ● DEC 2017

US Army Research Laboratory

Source-Code Stylometry Improvements in Python

by Gregory Shearer
ICF, Fairfax, VA

Frederica Nelson
Computational Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
December 2017	Technical Note	15 September 2017–31 October 2017

4. TITLE AND SUBTITLE
Source-Code Stylometry Improvements in Python

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Gregory Shearer and Frederica Nelson

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
US Army Research Laboratory
Computation Information Sciences Directorate(ATTN: RDRL-CIN-D)
Aberdeen Proving Ground, MD 21005

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TN-0860

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This technical note covers the work in rewriting existing source-code stylometry software into Python, and describes improvements to performance and maintainability and validation of results. Source-code stylometry is the process of attributing the authorship of source-code samples based on lexical, layout, and syntactic features extracted from code using machine-learning techniques, specifically random forest classifiers. The original work was conducted as part of a collaboration between the US Army Research Laboratory and Drexel University.

15. SUBJECT TERMS
source code, stylometry, attribution, machine learning, random forests, Python

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Gregory Shearer
Unclassified	Unclassified	Unclassified	UU	18	19b. TELEPHONE NUMBER (Include area code) (301) 394-4617

Contents

List of Figures

List of Tables

1. Introduction

Code stylometry is a means of authorship attribution for source or binary code. Much like a person can be identified via their handwriting or an author identified by their style or prose, programmers can be identified by their code. Provided a labelled training set of code samples (example in Fig. 1), the techniques used in stylometry can identify the author of a piece of code or even a compiled binary by utilizing the underlying structure of the code contained in the abstract syntax tree (Fig. 2) produced by the code. This method of attribution does not rely on author comments or whitespace features, and thus the features cannot be easily obfuscated to protect the code author from de-anonymization. Furthermore, by recreating the abstract syntax tree of compiled code using forensic processes, even compiled binaries can be evaluated for characteristics of code structure.

```
int foo()
{
   if((x < 0) || x > MAX)
      return -1;

   int ret = bar(x);
   if(ret != 0)
      return -1;
   else
   return 1;
}
```

Fig. 1 Sample code listing from code-stylometry paper made possible by a US Army Research Office grant (Caliskan-Islam et al. 2015)

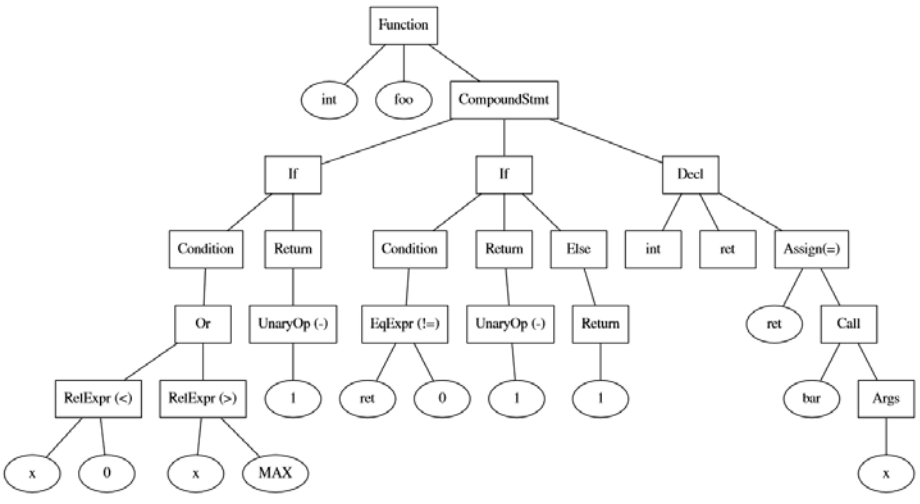

Fig. 2 Corresponding abstract syntax tree from de-anonymizing programmers' paper (Caliskan-Islam et al. 2015)

Stylometry research has proven that anonymous code contributors can be de-anonymized to reveal the original author, provided the author has published code before. This potential for de-anonymization must be considered both a tool and a threat, as stylometry is a technique that could be used by both friend and foe. As a tool, stylometry may be useful for identifying code contributions, including potentially identifying malware authorship. From an adversarial perspective, techniques to mitigate de-anonymization should be studied to reduce the risk to friendly authors. Other potential uses beyond de-anonymization include ghostwriting detection, software forensics, copyright investigation, and authorship verification.

Previous collaboration between Drexel University's Privacy and Security Laboratory and the Network Security Branch (NSB) of the US Army Research Laboratory (ARL) produced a number of published papers, informed research and transition efforts, and in general contributed to moving forward the state of the art. This prior work has demonstrated the overall feasibility of the technique, showing greater than 95% accuracy when attributing code from 1 author out of 250 (see Fig. 3) and accuracy greater than 90% identifying code authorship from a domain of 1600 authors in experimental datasets (Caliskan-Islam et al. 2015). In general, a larger author set will reduce accuracy while a smaller author set will increase accuracy. Inversely, more code samples per author increases accuracy, while fewer code samples per author decreases accuracy.

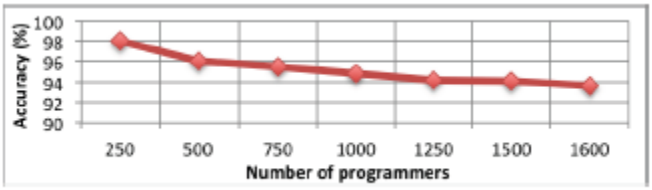

Fig. 3 Large-scale de-anonymization of 250–1600 code authors (Caliskan-Islam et al. 2015)

These results are for the closed-world case; that is to say, picking 1 author out of a known complete set. However, expanded open-world classification and multiauthor classification have also been examined (Caliskan-Islam et al. 2015; Stolerman et al. 2014). The research has been further expanded by considering binary stylometry (Rosenblum et al. 2011) with positive results from compiled code and encouraging results even when predicting attribution from obfuscating compiler-compiled code (Caliskan-Islam et al. 2015).

Table 1 Effect of obfuscation on de-anonymization from code-stylometry baseline paper (Caliskan-Islam et al. 2015)

Obfuscator	Programmers	Language	Results w/o obfuscation	Results w/ obfuscation
Stunnix	20	C++	98.89%	100.00%
Stunnix	20	C++	98.89%[a]	98.89%[a]
Tigress	20	C	93.65%	58.33%
Tigress	20	C	95.91%[a]	67.22%[a]

[a] Information gain features

2. Motivation

During collaboration between Drexel University and ARL, data processing for code stylometry has primarily been conducted by Drexel personnel on Drexel computers while learning and analysis research was conducted collaboratively by both parties. Thus, data-processing scripts resided primarily on Drexel hardware. Recently, a requirement emerged to transition or develop an ARL internal code-stylometry environment to demonstrate, share, and enhance or build upon the current state of the art in code stylometry to continue research and perform operational evaluations both individually and in conjunction with other available tools. In doing so, and to create the stand-alone environment, the entire process of code stylometry must be integrated, including data processing and learning and analysis.

The first path examined was a transition of code from the existing processing framework onto ARL systems. This initially was considered the fastest and easiest path. However, a number of compatibility issues were discovered during the effort to transition existing code, including significant challenges in finding obsolete versions of needed software dependencies, performance issues, and low readability of the research code. As a result of these challenges, the possibility was examined of simplifying and rewriting the code-stylometry software in Python on an ARL platform. Because ARL collaborates with Drexel University in researching code stylometry, the intent is to share the Python stylometry software with Drexel once the initial development is completed and provide updates as necessary as the project progresses.

3. Purpose

The new code-stylometry software aims to preserve the functionality of the original software while accomplishing the objectives of increased performance, increased readability, and better compatibility with existing operational and research platforms. All newly written code is in the Python programming language to

improve readability and interoperability. The number of dependencies required by the tool has been reduced from 5 to 3 for source-code stylometry, improving portability and ease of maintenance. All dependencies on nonpublically available code have been removed. Performance in terms of data-processing time has been improved by an estimated factor of 5 to 10 times by using a single initialization of the database server rather than multiple initializations throughout the experiment, reducing the amount of time required to process code.

The aim of the new software is to act as a base for new ARL internal research on code stylometry and facilitate greater control over potential updates, patches, and upgrades to the software. An independent codebase will allow greater flexibility in designing experiments and enhanced interoperability with other applications as needed. Additionally, the reduced dependency set and more interoperable design allows for easier installation on computers within the ARL environment. Transitions to other ARL branches or other organizations should also be significantly easier with the new software compared to the old software.

4. Tool Components

The code-stylometry tool is composed of several parts that constitute a general workflow for code or binary processing, feature extraction, and learning/prediction.

4.1 Dataset Definition

The first step of the tool workflow is the extraction of samples from a labelled dataset. Notionally, this extraction could include the entire dataset or any smaller portion of it. In the current build of the Python version of the stylometry tool, the extraction is handled via a script that accepts as input constraints on which and/or how many authors and/or files should be drawn from a larger dataset for use in a smaller subset of the dataset.

The required dataset format is indexed as follows:

1) Dataset main directory
    a. Author directories
        i. Individual files

The script iterates through the dataset's top-level directory and selects a subset of authors that matches the input criteria for number of authors and required problems per author. It then creates a new directory in the same format as the original dataset directory, containing only the authors and problems of interest.

4.2 Feature Extraction

The next step is to create a feature set from all of the samples within the given dataset. The feature set must capture enough information about the code to be informative in terms of authorship attribution, but should not be so large as to make machine learning on the feature set computationally infeasible.

In general, there are 3 types of features that can be drawn from source code: 1) *Lexical* features deal with the word vocabulary used in the source code, 2) *layout* features deal with whitespace formatting, and 3) *syntactic* features are drawn from the layout and content of the abstract syntax tree. Syntactic features are the most resilient to obfuscation, whereas layout can be trivially altered and both layout and lexical features do not survive the compilation process. For lexical and layout features, the source code is read directly and processed by a function within the processing script. For syntactic features related to the abstract syntax tree, a more complex process is needed to parse the code. Both the original research code and this Python implementation use a tool called "Joern", a fuzzy parser designed specifically for processing code that may be incomplete (Yamaguchi et al. 2014).

Joern inserts the abstract syntax-tree layout of the code sample set into a "Neo4j" graph database. In essence, the structure of the graph in the database is a large tree, with a root node of the main data directory, ascending through author directory and, finally, an abstract syntax tree for each individual problem. The abstract syntax tree (as shown in Fig. 2) decomposes complex operations into smaller parts, finally resulting in leaf nodes. This syntactic structure information can be drawn from the Neo4j database through queries using the Python library "Py2Neo" as an interface.

For the validation experiment later in this technical note (Section 6), we used the following:

- source-code word unigram's term frequency,
- source-code word unigram's average position within the document (measured as 0 at start, 1 at end),
- abstract syntax-tree-node types' term frequency,
- abstract syntax-tree-node types' average depth (depth within abstract syntax tree hierarchy), and
- abstract syntax-tree-node bigrams' term frequency.

4.3 Feature Mapping

Next, the processing script maps features on a per-sample basis to form a feature-to-sample mapping. Each code sample has feature information collected from both

the original source file (for lexical and layout features, if any) and the corresponding abstract syntax tree for that sample file (for syntactic features). The unique features for each sample are also added to an experiment-wide corpus aggregator. This aggregator collects all unique features from each sample to create a single, unified, experiment feature set that includes all features arising from samples in the dataset. This unified feature set will be used to create and format the data for the machine-learning model.

The experiment feature set can be reduced depending on experiment parameters. For example, features that appear only once in a corpus are useless for prediction, as these features could never correlate 2 separate samples within the corpus; accordingly, in the validation experiment in Section 6, the final feature set includes only features that appeared more than once in the overall corpus.

After feature reduction, the samples are assigned a feature data array based on which features they possess from the overall corpus-feature dataset. Features from the corpus-feature set that do not exist in the sample features are filled in with zeros in the feature data for each sample.

4.4 Learning and Prediction

Finally, the script applies a random forest classifier—implemented in the Python machine-learning library's *Scikit-learn* (Pedregosa et al. 2011)—to create a model of how the mapped features per sample are associated with author labels. Once a model has been created, the script can attempt to predict the authorship of new samples whose authorship is unknown, provided we know the author is within the known set of authors. So long as the features used for stylometry are informative as to authorship attribution, prediction of unknown authors within a set should be possible. We can validate the results of the prediction using a technique called cross-validation (discussed in Section 6).

The random forest classifier is an ensemble learner built from a collection of decision trees. Each decision tree is created by randomly sampling training samples with replacement from the sample set. During classification, each test example is classified by each of the trained decision trees and the results are subsequently aggregated. In essence, the trees "vote" on the overall classification of each sample, with the eventual label being the most popular classification from the individual trees. The random forest model as used by the learning script uses 500 decision trees as estimators to form an ensemble classifier.

5. Specific improvements

Compared to the original research code in which the Joern tool was run on each file individually, only 1 instantiation of Joern is needed to process all of the sample files using the new methodology. This reduces the amount of time needed for processing, and allows for a single Neo4j database to be used rather than repeatedly creating new databases.

Similar to Improvement 1, the new methodology only requires 1 instantiation of Neo4j during the entire experiment, rather than 1 instantiation per file. This is achieved by reading in all Joern data at the beginning of the experiment, then searching for specific subtrees within the database as the author files are iterated over. This vastly reduces the amount of time needed for server startup and shutdown.

The Python source-code stylometry rewrite reduced the number of dependencies from 5 to 3. This was accomplished by using the native functionality of Py2Neo to read from the Neo4j database rather than a collection of 3rd-party Python and shell scripts. In addition, the dependencies it does require are the newest public versions of the dependencies, rather than old versions. This should make the code more flexible and more easily movable to different platforms.

The new data-processing methodology preserves dataset integrity by not writing any new files to the experimental dataset. The research code wrote several files to the dataset for each sample problem examined. This ensures the dataset has not been altered in any way and can be used again without cleanup for subsequent experiments with new processing methods. It also reduces the file input/output overhead, leading to reduced time to process files.

The new Python stylometry code significantly streamlines the overall codebase by deleting out-of-date functions and functions that may have been used for research but are now deprecated. Only 3 key scripts are required for the entire Python source-code stylometry workflow. This should help ensure the codebase is more readable and more easily maintained.

The migration to Python language scripts improves compatibility with the ARL environment. We faced many challenges setting up an environment with the requisite dependencies of the research code. This upgrade to code stylometry should not only serve as an enhancement but ensures future research and development on stylometry integrates more smoothly with other efforts.

6. Validation

As with the original research in code stylometry, the validation experiments for this work will use the Google Code Jam dataset, a collection of C, C++, and Python source-code samples labelled by author and problem number. This dataset originates from the Google Code Jam challenge, a programming competition to write code to solve a series of programming problems.

All accuracy evaluations for both the original work and new work use the k-fold cross-validation method, where data were split into training and test sets stratified by class (in this case, author). The number of code samples per author in the training and test sets was identical for all authors. The parameter k is the number of segments the data are split into, with k-1 segments used for training and the remaining segment tested upon.

A baseline demonstration used earlier in the stylometry project uses a small subset of the larger Google Code Jam dataset containing 10 specified authors with 9 files each. Using this dataset, 9-fold cross validation is performed to obtain a score from the machine-learning classifier, meaning that for each author class 8 samples will be used for training the model and 1 sample will be used for testing. This process will be repeated until all combinations (or folds) of the dataset have been tested.

The feature set for the original and Python stylometry versions is different as are the exact parameters of the learning mechanism. Thus, results are not anticipated to be exactly the same; however, they should be similar enough for comparison. For the initial validation, we will only examine closed-world source-code stylometry rather than binary stylometry or open-world attribution situations.

In testing, the original code stylometry demo obtained a 9-fold cross-validation accuracy of 92.9% averaged over 5 runs. (See Table 2.) The new code-stylometry tool using the same data obtained a 9-fold cross-validation accuracy of 93.6% averaged over 5 runs. This suggests that for the basic authorship-attribution task, the Python-based implementation is capable of achieving comparable results.

To further validate the reproduction of results, we used a dataset of 250 authors who had completed at least 9 problems from the 2012 Google Code Jam problem set. Each author has exactly 9 of their completed problems from the 2012 problem set assigned to them as samples. From the de-anonymizing programmers' paper the best result obtained for this dataset was 96.83% (Caliskan-Islam et al. 2015, table 5) after information gain was applied. The average (3 repeated runs) result from the Python version of the stylometry tool using the same data without information gain features was 96.77%.

Table 2 Results of the validation experiment; time to complete depends on the hardware used to run the processing

	10 authors' 9 files (demo dataset)	250 authors' 9 files CPP[a] 2012 from Google Code Jam dataset	250 authors' 9 files CPP[a] 2014 from Google Code Jam dataset
Target 9-fold cross-validation accuracy (from Caliskan-Islam et al. 2015)	92.9%	96.83%	95.07%
Python 9-fold stratified cross-validation accuracy	**93.6%**	**96.77%**	**97.56%**
Time to create experiment dataset	1.0 s	~12.0 s	~12.0 s
Time to process code and extract features	548.8 s (9 min 8.8 s)	17151 s (4 h 45 min 51 s)	17739 s (4 h 55 min 39 s)
Time to run learning and evaluation	24.8 s	3587.1 s (59 min 47.1 s)	3503.9 s (58 min 23.9 s)

[a] C Plus: C++

It appears 9-fold cross-validation accuracy is similar between the original work and the Python implementation. This is somewhat surprising considering the relative simplicity of the initial feature set used by the Python implementation for source-code stylometry. The original paper (Caliskan-Islam et al. 2015) remarks that in many cases the abstract syntax-tree bigram's features are enough to achieve similar results to the full feature set. It may be the case that for these datasets this simpler feature set is sufficient. We also note the random forest model used for this validation experiment used a higher number of classifiers, 500 rather than 300, which may have allowed for more precise model creation.

7. Conclusion

It is intended that the improvements listed here to the code-stylometry tool will be used by ARL researchers to further research in the code-stylometry field. Additionally, the tool may form the basis of an operational prototype for code stylometry. By rewriting the tool in Python and revising the workflow for certain aspects of data processing, we obtained a 5–80-times reduction in data-processing time, reduced the number of dependencies for source-code stylometry from 5 to 3, reduced the number of external dependencies, and streamlined the existing research code into a straightforward tool. We anticipate continued development to integrate features such as attribution from binaries via binary disassembly and recompilation, open-world case handling, and multi-author handling.

8. References

Caliskan-Islam A, Harang R, Liu A, Narayanan A, Voss C, Yamaguchi F, Greenstadt R. De-anonymizing programmers via code stylometry. Proceedings of the 24th USENIX Security Symposium; 2015 Aug 12–15; Washington (DC). USENIX.

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, et al. Scikit-learn: machine learning in Python. J Machine Learn Res. 2011;12:2825–2830.

Rosenblum NE, Zhu X, Miller BP. Who wrote this code? Identifying the authors of program binaries. Proceedings of the Computer Security ESORICS 2011; 2011 Sep 12–14; Leuven, Belgium. p. 172–189.

Stolerman A, Overdorf R, Afroz S, Greenstadt R. Classify, but verify: breaking the closed-world assumption in stylometric authorship attribution. Presented at: 10th IFIP Working Group 11.9 on Digital Forensics. International Federation for Information Processing; 2014 Jan; Philadelphia, PA.

Yamaguchi F, Golde N, Arp D, Rieck K. Modeling and discovering vulnerabilities with code property graphs. Proceedings of IEEE Symposium on Security and Privacy (S&P); 2014 May 18–21; Washington (DC). IEEE Computer Society; p. 590-604.

List of Symbols, Abbreviations, and Acronyms

ARL	US Army Research Laboratory
CPP	C Plus, C++
NSB	Network Security Branch

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
2 (PDF)	DIR ARL IMAL HRA RECORDS MGMT RDRL DCL TECH LIB
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
2 (PDF)	DIR ARL RDRL CIN D G SHEARER F NELSON