Python Source Code De-Anonymization Using Nested Bigrams

Conference Paper · November 2018

DOI: 10.1109/ICDMW.2018.00011

3 authors:

Pegah Hozhabrierdi
Syracuse University
2 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Dunai Fuentes
École Polytechnique Fédérale de Lausanne
1 PUBLICATION **0** CITATIONS
SEE PROFILE

Chilukuri Mohan
Syracuse University
34 PUBLICATIONS **97** CITATIONS
SEE PROFILE

Python Source Code De-Anonymization Using Nested Bigrams

Pegah Hozhabrierdi
EECS Department
Syracuse University
Syracuse, USA
phozhabr@syr.edu

Dunai Fuentes Hitos
Matilock, Inc.
Huelva, Spain
dunai.ianud@gmail.com

Chilukuri K. Mohan
EECS Department
Syracuse University
Syracuse, USA
mohan@syr.edu

***Abstract*—An important issue in cybersecurity is the insertion or modification of code by individuals other than the original authors of the code. This motivates research on authorship attribution of unknown source code. We have addressed the deficiencies of previously used feature extraction methods and propose a novel approach: *Nested Bigrams*. Such features are easy to extract and carry substantial information about the interconnections between the nodes of the abstract syntax tree. We also show that for large number of authors, a *Strongly Regularized Feed-forward Neural Network* outperforms the Random Forest Classifier used in many code stylometric studies. A new ranking system for reducing the number of features is also proposed, and experiments show that this approach can reduce the feature set to 98 nested bigrams while maintaining a classification accuracy above 90 percent.**

***Index Terms*—source code de-anonymization, source code stylometry, abstract syntax tree, feature extraction, feature ranking**

I. Introduction

Stylistic analysis has long been used to answer questions concerning the authorship of articles and books. Recent research has attempted to determine whether a similar analysis can be applied to computer source code for authorship attribution. This has considerable application to the field of cybersecurity, in identifying whether a small part of a computer program may have been authored with malicious intent by someone other than the main author. In addition to the applications in security (e.g., malware authorship [1]) and copyright (e.g., plagiarism detection [2]), source code authorship attribution addresses well-known challenges in software engineering such as code clone detection [3].

Over time, improvements have been made in identifying source code authorship, from Oman and Cook's [4] use of formatting features to the extensive study of Caliskan-Islam *et al.* [5], using a more comprehensive feature-set (including lexical, layout and syntactic features). Our study is motivated by several potential improvements and extensions that remain to be investigated:

- Many of the previous studies focus on C/C++ and Java samples due to the abundance of the available data. Some methods (*e.g.,* [5]) achieve high accuracy levels on such code, but do not generalize well to other languages.Thanks to the steady increase in the usage of Python since 2012, a substantial amount of data is now available so that we can examine Python code stylometric features. A few recent studies have tackled this issue [5], [6], but the performance of the Python code classifier in [5] is relatively poor, while [6] uses a very complex model.
- Previous methods often used thousands of features for training the classifiers that leads to over-fitting, motivating feature set reduction techniques. The previous use of information gain to reduce the number of features has lead to significantly worse performance, for Python code as in [5].
- Regarding the usage of the information gain criterion, several questions remain unexplored, such as: (1) What is the appropriate threshold of information gain (to capture the key features)? (2) Does information gain truly capture the hidden "signature" in each code? (3) How does it compare to other commonly used methods?

In this study, we address the above issues and obtain insight into useful properties of the feature-set. We have relied on data available from Google Code Jam (GCJ) annual competitions, facilitating a fair and meaningful comparison with the previous stylometric studies of Python source code that used the same dataset.

We begin with a brief review of the literature in Section II. The proposed features (nested bigrams) are introduced in Section III. In Section IV-A, we show our classification results using only the AST nested bigrams as the feature-set, experimenting with both random forest and neural network classifiers. The results are compared to previous studies (Section IV-B). Next, we address feature ranking in Section IV-C, and examine two measures: information gain ranking and correlation-based ranking. Section V summarizes our conclusions.

II. Related Work

Author de-anonymization has been mostly studied from the perspective of plagiarism detection, as reviewed in [7] and [6]. Advances in this field are mostly owing to improvements in the features used for author identification. Hence we categorize these studies by their selected features:

Layout features. Pioneering studies ([4] and [8]) focus on extracting the *format* of the source codes; but these features are prone to error and vulnerable to obfuscation attacks.

Lexical features. These include quantitative variables such as the number of occurrences of specific keywords, proportion of lines that include comments, the average length of identifiers, number of strings, chars and literals, *etc.* These features combined with *n-grams* have been used for author de-anonymization [2], [9], [10].

Syntactic features. The vulnerability of layout and lexical features to obfuscation attacks motivated the attempts towards a more robust set of features. Pellin [11] introduced a new set of features obtained by parsing the abstract syntax tree of the source code, further improved by Caliskan-Islam *et al.* [5]. Such features depend only on the syntax of the code and are more resistant to attack. Height of the AST, n-grams of the nodes, the depth of the tree and the number of leaf nodes are examples of syntactic features. Alsulami et al. [6] use an LSTM-based neural network to extract informative features from AST. In their method, they preserve the information between two nodes that don't have immediate parent-child relationship.

III. Feature Extraction

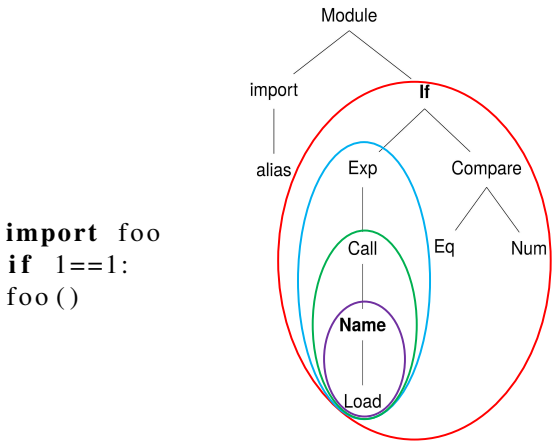
The main contribution of our work is to introduce an efficient and smaller feature-set that captures the signature of an author. *Nested Bigrams* retain information contained in the relationships between non-adjacent nodes, with the *nesting* of subtrees. To take the non-immediate relationships (siblings and non-immediate descendants) into account, we associate each node with the information of its descendants. Nested bigrams can be extracted rapidly while retaining long-distance connection information (throughout the AST) without the use of complicated encoding mechanisms. Our experiments show that these bigrams (tuples) preserve author signatures as well as or better than the representations learned by the LSTM and biLSTM encoders.

Figure 1 illustrates a code snippet and its corresponding abstract syntax tree. Each of the circles drawn in Figure 1 represents the information of the top most node in that circle (only four nodes are depicted by circles for clarity). For example, the `If` node carries all the information of all the nodes enclosed in the red circle, expressed as follows:

```
If(test=Compare(left=Num(n=1),ops[Eq()],
   comparators=[Num(n=1)]),body[Exp(
   value=Call(func=Name(id='foo',
   ctx=Load()),args=[],keywords=[],
   starargs=None,kwargs=None))],orelse=[])
```

All the nodes within the red circle (i.e. `Compare, Eq, Num, Exp, Call, Name, Load`) are stored as the parameters of the `If` node. The blue circle shows the information held by the `Exp` node (i.e. the nodes `Call, Name, Load`):

```
Exp(value=Call(func=Name(id='foo',
      ctx=Load()), args=[], keywords=[],
      starargs=None, kwargs=None))
```



(a) (b)

Fig. 1: The abstract syntax tree of a Python code snippet. (a) Code snippet (b) The corresponding AST. Each circle represents the information stored by one node (the top most node in each circle circle).

The same holds for the green and purple circles as well. Each circle represents a *node* in our AST. The connection of `If` and `Exp` nodes (the red and blue circles) is a sample of our nested bigrams. Relationships are preserved between ancestors and descendants (*e.g.,* `If` and `Name`), as well as siblings (*e.g.,* `Eq` and `Name`).

IV. Classification

In this section, we introduce the structure of a Strongly Regularized Feed-forward Neural Network (SRFNN) and evaluate its performance when trained with nested bigram features.

Data. Following the previous studies in this field, we choose Google Code Jam (GCJ) dataset[1] to compare the results. GCJ is an annual programming competition and consists of several rounds of questions. The participants may submit their answers in any language. The questions and the submitted codes are made available in their website.

Features. The AST was extracted with the help of the *ast* built-in module in Python 2.7 [2]. Only the nested bigrams used by at least two users are selected to ensure that the classifier is not biased towards the specific names of variables used by only one user. Nested bigrams extracted from the (AST) of a python source code are one-hot encoded and multiplied by the frequency of appearance. To facilitate future research, all our source code is publicly available online[3].

A. SRFNN vs. RFC

[5] and [6] used data with 9-10 code samples for each author. We have experimented with different number of code samples, and have found that even with only three code

[1] https://code.google.com/codejam/past-contests

[2] We choose Python 2.7 rather than 3.x as our platform because our dataset contains source code as old as 2012 and mostly written in Python 2.x.

[3] https://github.com/Pegayus/nested-bigrams

Layers	Output Size
Dropout (0.5)	number of features
Fully-connected	500
ReLU	500
Batch-normalization	500
Dropout (0.5)	500
Fully-connected	number of classes
Softmax	number of classes

TABLE I: SRFNN Architecture

samples per user, we can achieve higher accuracy with SRFNN than the previous studies while increasing the number of classes. For both classifiers, we use stratified *3*-fold cross-validation for the evaluation of the training model; *i.e.*, for each user, we train with only two code samples and evaluate on the third. We report the average accuracy across all folds.

RFC. By tuning the hyper-parameters of this model among all our experiments, we fix the number of trees to 500 and let the trees grow to their maximum height (our tuning results showed that limiting the height of the trees has negligible impact on the accuracy). We choose entropy as splitting criterion, to keep the model closer to that of previous studies; similar results were obtained with Gini.

SRFNN. Our target tasks train on as few as two code samples per user (testing on a third), with up to 7099 non-user specific or problem-specific features per code sample. The scarcity of training data per class makes the tasks comparable to applications such as Omniglot [12] or Mini-Imagenet [13]. Instead of using the complex procedures proposed in such literature, we regularize our network and training procedure to generalize effectively from a few examples. We use Dropout [14] to model the fact that a user's style may be composed of multiple stylometric indicators that may appear independently. We find that randomly dropping up to 75% of the input features helps to avoid over-fitting, forcing our model to learn the multiplicity of informative features. To prevent over-fitting, our network architecture is kept simple, as shown in Table I. Finally, our training schedule consists of a sequence of steps with progressively decreasing learning rate, with warm restarts [15], and paired decreasing weight decay [16], which helps us regularize parameters while maintaining low bias towards the end of the training phase. They decrease from 0.1 to 10^{-5} and from 0.1 to 10^{-7} respectively. We train with small batches of size 20, preserving high stochasticity.

Experiments. Our optimizer of choice is Adam [17], and our implementation uses the PyTorch[4] platform. We have used two different splits of the data: (1) users who have solved a mixture of different and similar problems, and (2) those who have the same set of problems. In case of the mixed problem-sets, the classifier has an advantage as more information is provided (the difference in the coding style and the difference in the problem addressed). The nested bigrams are extracted from all codes in the dataset. Then, for each source code, the normalized frequencies are calculated (the frequency of each bigram is divided by the length of the source code).

[4]https://pytorch.org/

Discussion. Table II compares the performance of RFC and SRFNN on different datasets. The results of the first column can be compared to those of [5]: for 229 users with nine codes each who solved different problems, their accuracy was 53.91%, and for 23 users who solved the same problems, they achieved 87.93%. Considering the higher difficulty of classification for the same set of problems, their higher accuracy for users with same problem-set can only mean that the number of classes (users) has had a powerful impact.

The largest set of users who solved the same set of problems in our dataset consist of 249 users with three codes each. The total number of users with three codes was 478. The second and fourth columns of Table II show the accuracies for these two groups. To compare the performance of the classifiers for the mixed and same set of problems, we also train on a subset of 249 users with three codes each who have solved different problems.

Comparing the third and fourth column of the table, the RFC appears to capture the difference in the problem-sets than the difference in the style of coding (nine percent decrease in accuracy) while SRFNN does the opposite (five percent increase in accuracy). Adding the second column to this observation, RFC appears to be more sensitive to the number of classes than SRFNN. To study the effect of (1) the number of users (classes) and (2) the number of codes per user on the accuracy in a more systematic way, we repeated the training on subsets of the users who solved the same three problems (cf. Figure 2). We also used a dataset of 13 users who have exactly six problems in common, with results shown in Figure 3.

	mix_81u_9c (7,269)	mix_478u_3c (14,480)	mix_249u_3c (7,389)	same_249u_3c (7,098)
RFC	**97.67%**	66.67%	72.30%	65.19%
SRFNN	95.6%	**71.6%**	**73.5%**	**77.5%**

TABLE II: Classification accuracies of RFC vs. SRFNN using nested bigrams. From left to right, the meaning of the columns are: {81, 478, 249, 249} users with {9, 3, 3, 3} codes each and {mixed, mixed, mixed, same} set of problems. The numbers in parentheses show the number of extracted features(bigrams) used by at least two users.

Figures 2 and 3 capture the effects of the number of authors (classes) and the number of codes per author. The points are the average of the cross-validation accuracies and the standard deviation of the results is overlaid. Together, Table II and the two figures suggest a tradeoff between these two parameters. It is clear that SRFNN surpasses RFC when the number of users is large, with a sharp decrease of accuracy in RFC from 20 users to 200 users in Figure 2. However, Figure 3 suggests that for a few code samples per author, RFC performs better than SRFNN. For users with five or more code samples, this difference is negligible. As shown, for two codes per author, SRFNN seems to be performing slightly better. This can be due to the effect of the small number of users (classes). Here, the impact of the number of classes has been stronger than that of number of codes per author. The same behavior is observed from Figure 2 in which for the small number of codes per user (three) and smaller number of classes (20), the effect of the number of classes appears to be greater, and

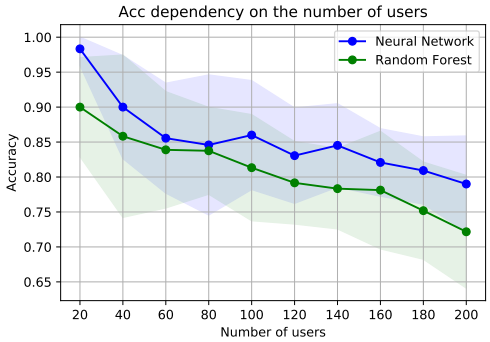


Fig. 2: The classification accuracy (Acc) for different class sizes. Each user has 3 code samples.

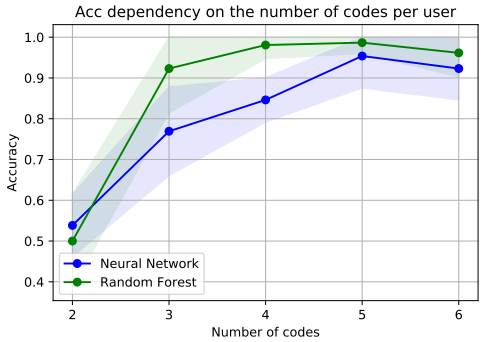


Fig. 3: The classification accuracy (Acc) for different number of codes per user for 13 users.

	Small C	Large C
Small A	RFC	RFC/SRFNN
Large A	SRFNN	SRFNN

TABLE III: Choice of better classifier, depending on the number of authors (A) and the number of codes per author (C)

SRFNN outperforms RFC by a large margin.

Summary. These results can be summarized as follows:

- RFC is more sensitive to the type of problems a code aims to solve, but cannot learn the full difference in the coding style of the programmers and is not a good option for classifying authors who solve the same set of problems. SRFNN is a better classifier for such problems.
- There is a tradeoff between the number of classes and the number of sample codes per user. RFC tends to perform worse for the larger number of users while it does a better job for the users with smaller number of code samples, as shown in Table III.
- The SRFNN performs significantly better for large number of classes and large number of codes per author, suggesting higher *generalization capacity* to accommodate a diversity of styles than the RFC. This paves the way to even larger classifiers to be used on massive datasets.

B. Comparison with Previous Results

Among the literature on author de-anonymization so far, only [5] and [6] have offered competitive results for Python source code. The feature-sets in these two studies consist of all syntactic features drawn from AST. We train our two classifiers, RFC and SRFNN, on the same number of authors with mixed/same set of problems, using nested bigrams. Table IV compares the result of our experiments with the two other works. We executed each classifier five times and reported the average of the accuracies. We also repeated the experiments with fewer code samples per user. Therefore, our classifiers have the disadvantage of less training data to capture the coding styles. For each problem, [5] uses nine and [6] uses 10 different coding styles. We only used four for the same number of authors and, still, obtained a better accuracy.

Table IVa illustrates the superiority of the nested bigrams over the code stylometric features used by [5]. Table IVb

	Features	23 authors [same]	229 authors [mixed]
RFC [5]	all AST	87.93 [9 codes]	53.91 [9 codes]
RFC [this work]	AST nested bigrams	**93.47 [4 codes]**	83.29 [4 codes]
SRFNN [this work]	AST nested bigrams	88.04 [4 codes]	**86.20 [4 codes]**

(a)

	Features	25 authors [same]	70 authors [same]
LSTM [6]	all AST	92.00 [10 codes]	86.36 [10 codes]
biLSTM [6]	all AST	**96.00 [10 codes]**	88.86 [10 codes]
RFC [6]	all AST	86.00 [10 codes]	72.90 [10 codes]
RFC [this work]	AST nested bigrams	**95.56 [4 codes]**	**89.35 [4 codes]**
SRFNN [this work]	AST nested bigrams	88.00 [4 codes]	86.40 [4 codes]

(b)

TABLE IV: Comparison with existing literature: "all AST" means that all the possible features for AST have been used (such as height of the tree, count on leaf nodes, n-grams, etc.) (a) Comparison with Caliskan et al. [5]. (b) Comparison with Alsulami et al. [6].

shows that with fewer training data, nested bigrams can surpass LSTM based networks for larger number of classes, with similar results for smaller datasets. Another interesting observation is the slope of decrease in accuracy from 25 users to 70 users. LSTM's and biLSTM's accuracies are decreased by 6.1% and 7.43% respectively while this number for RFC and SRFNN is 6% and 1.8% accordingly. This demonstrates the robustness of SRFNN model with increases in the number of classes.

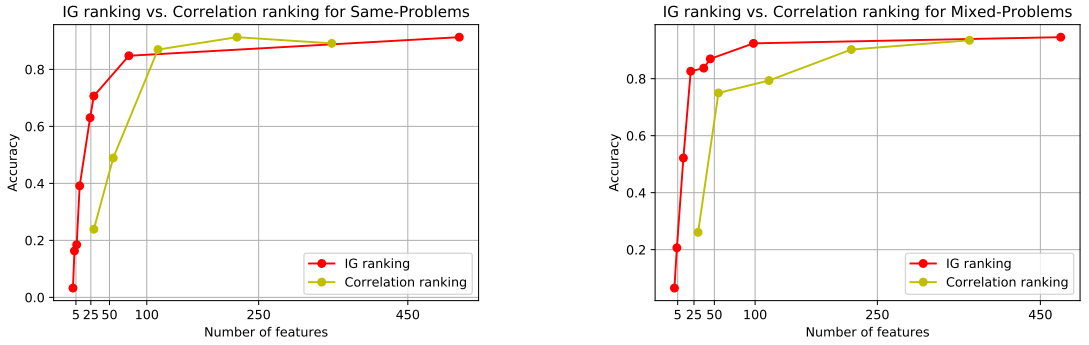
C. Feature Ranking

The major differences between our study and [5] are the type and the size of the feature-set. Their initial feature-set contains thousands of redundant lexical, layout and syntactic features (120,000 for their C/C++ experiments) which result in over-fitting. Their approach for extracting more informative feature-set was to choose those features with positive information gain. They reduced the number of their C/C++ features to 928 from 120,000 and improved the result of their classification. Use of their selection criteria had little effect on the results with nested bigrams; for example, in one of our experiments, it reduced the size of our features from 1,830 to 1,829, with no change in accuracy.

In our experiments, decreasing the number of our nested bigrams via information gain decreased the classification accuracy, with a slow drop. However, if we don't seek the

	Accuracy > 80%				Accuracy > 90%			
Ranking Methods	optimized threshold	number of features	accuracy(%)	compression(%)	optimized threshold	number of features	accuracy(%)	compression(%)
IG[mix]	1.2	**21**	83.70	97.38	0.5	**98**	92.39	87.78
Corr[mix]	0.5	117	82.62	85.41	0.7	218	90.22	72.82
IG[same]	0.5	**76**	82.61	90.88	0.1	519	91.30	37.77
Corr[same]	0.5	115	83.68	86.21	0.7	**221**	91.30	73.50

TABLE V: The results of feature pruning by two different metrics; information gain ranking and correlation based ranking. The first test captures the minimum feature-set by each of the ranking measures that gives accuracies above 80%. The second test is for accuracies above 90%.



(a) (b)

Fig. 4: The decrease in the size of feature-set affects the accuracy differently for IG and correlation based rankings. (a) shows this difference for the dataset of 23 authors with four sample codes per author who solved the same set of problems. (b) depicts the same results for the dataset of 23 authors with four codes and a set of different problems. The correlation thresholds are chosen from [0.1,0.3,0.5,0.9] with 0.1 giving the minimum results in the figures. That explains the sudden truncation of the correlation plots .

"best" possible accuracy and accept accuracies above a certain threshold, a very small subset of our features can suffice.

In Section IV-B, we compare our "best" possible results with the best achieved by other studies, using our complete feature-set. Here, we propose a *tuning* procedure which gives us the minimum set of nested bigrams that satisfy a minimum accuracy required. Our tuner is based on the information gain/-correlation of feature vectors. In the end, we also rank those selected features by the tuner through a *one-out* (or *leave-one-out*) ranking procedure. We also compared our results with a correlation based measure between the feature vectors.

Information Gain Ranking. The ranking of the features utilizes two steps: (1) Tuning the information gain threshold from which the minimal set of features satisfying a fixed accuracy threshold is obtained, and (2) Ranking the *importance* of those selected features by one-out approach. We choose 80% and 90% as our fixed accuracy thresholds. In the tuning process, we also investigate how accuracy varies with the number of features.

Tuning: For each threshold t of a set of information gain thresholds, we drop the features whose mutual information with the true classes is below t. Then, RFC is run through the dataset using the new features to get the accuracy. The RFC specifications are the same as in Section IV. Ultimately, we pick the t that gives the minimum set of features for an accuracy above a fixed threshold.

Ranking: All the pruned features are fed to a one-out ranking system that calculates the "importance-score" of each feature. The one-out approach removes one feature at a time, obtains RFC accuracy using all remaining features, and assigns $1 - accuracy$ as the score of that feature.

Correlation-Based Ranking. Here, out of the pairs of features with a higher correlation than the specified correlation threshold t, one is removed as redundant. The same tuning and ranking procedure as IG based ranking is adopted.

We employ the two ranking measures on two sets of data: (1) 23 authors solving a mixed set of problems with four sample codes per author, and (2) 23 authors solving the same set of problems with four sample codes per author. We look for the minimum feature-set that provides accuracy above 80% and 90%. The total number of features for dataset (1) and (2) are 802 and 834 respectively. As shown in Table V, 21 and 76 features are sufficient to obtain accuracy above 80 percent, with information gain ranking. The complete list of these features can be found in our repository. IG and Correlation measures differ in the resulting top selected features.

Discussion. IG ranking method extracts fewer features while giving higher accuracies than that of correlation based ranking. However, with a higher accuracy threshold, IG ranking has larger jumps in the number of features. The performance of correlation based ranking is comparable between the two datasets when IG ranking is more sensitive; IG performs considerably better for a dataset of different problems. The classification accuracy using the full feature-set is 93.45% (for mixed set) and 92.39% (for same set) respectively. Both the ranking methods get a comparable accuracy to that of the complete feature set with fewer features. To investigate the effect of the decrease in feature size on the accuracy, we generate the plots in Figure 4, with points generated during the tuning procedure of Table V. Correlation based measure, with more fluctuations, seems to be unpredictable for ranking purposes. The higher number of features does not necessarily mean higher accuracy by correlation ranking. However, IG shows a smooth increase and an actual *saturation point*.

Although higher accuracies can be achieved by correlation ranking measure, the sharp drop in accuracy for less than 100 features indicate the superiority of IG based ranking for the case of limited number of features. For achieving higher accuracies, correlation measure offers smaller steps in increasing the number of features whereas IG ranking suffers from the sudden leap in the number of features to achieve the same accuracy.

V. Conclusions

We have addressed several existing problems in source code de-anonymization. We have shown that Python source codes preserve the hidden signature of the authors in their *Nested Bigrams*. This shows the importance of syntactic features to evaluate author styles in Python source code. A Neural Network based classifier, SRFNN, was introduced to overcome the deficiencies of the Random Forest for large sets of authors. Comparison of our results with previous works indicates the superiority of our feature-set and the relative robustness of SRFNN. From an original feature space of more than 800 dimensions, our IG-Ranking method could capture, with 90% accuracy, the signature of 23 authors with only 98 nested bigrams. In future work, we plan to focus on evaluating our method against obfuscation attacks as well as new methods of feature reduction (e.g. compression methods) and interpretable embeddings.

Our main contributions can be summarized as follows:

1) We proposed the use of an unexplored encoding of the abstract syntax tree (AST) for our classifiers: *nested bigrams*. Nested bigrams hold the information between any two nodes that are vertically or horizontally connected in the tree, keeping long-distance relations available to the classifier. Because these features are extracted from the AST, they are also robust to obfuscation, as noted by [6], making them preferable to layout and lexical features.
2) Random Forest Classifiers (RFC) have been used in many recent studies on source code de-anonymization [5], [6]. However, our experiments indicated that the RFC results worsen rapidly as the number of classes increases. We proposed a Strongly Regularized Feed-forward Neural Network (SRFNN) model that can achieve higher accuracy than an RFC when the number of classes is large. We discuss conditions appropriate for the use of each of these classifiers.
3) Our feature-set, without information gain pruning, surpasses the previous results in [5], but leaves open the question of whether good performance can be obtained with fewer features. We introduced a method that shrinks the size of our feature-set by 88 percent and still retains the accuracy above 90 percent. We use an *information gain ranker* which gives us the top 98 features with high classification accuracy, outperforming a *correlation ranker*.

References

[1] S. Alrabaee, P. Shirani, M. Debbabi, and L. Wang, "On the feasibility of malware authorship attribution," in *International Symposium on Foundations and Practice of Security*, pp. 256–272, Springer, 2016.

[2] S. Burrows and S. M. Tahaghoghi, "Source code authorship attribution using n-grams," in *Proceedings of the Twelth Australasian Document Computing Symposium, Melbourne, Australia, RMIT University*, pp. 32–39, Citeseer, 2007.

[3] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of computer programming*, vol. 74, no. 7, pp. 470–495, 2009.

[4] P. W. Oman and C. R. Cook, "Programming style authorship analysis," in *Proceedings of the 17th conference on ACM Annual Computer Science Conference*, pp. 320–326, ACM, 1989.

[5] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry," in *24th USENIX Security Symposium (USENIX Security), Washington, DC*, 2015.

[6] B. Alsulami, E. Dauber, R. Harang, S. Mancoridis, and R. Greenstadt, "Source code authorship attribution using long short-term memory based networks," in *European Symposium on Research in Computer Security*, pp. 65–82, Springer, 2017.

[7] O. Mirza and M. Joy, "Style analysis for source code plagiarism detection," in *Plagiarism across Europe and Beyond: Conference Proceedings. Brno: MENDELU Publishing Centre*, pp. 53–61, 2015.

[8] E. H. Spafford and S. A. Weeber, "Software forensics: Can we track code to its authors?," 1992.

[9] R. Kilgour, A. Gray, P. Sallis, and S. MacDonell, "A fuzzy logic approach to computer software source code authorship analysis," 1998.

[10] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan, "N-gram-based detection of new malicious code," in *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, vol. 2, pp. 41–42, IEEE, 2004.

[11] B. N. Pellin, "Using classification techniques to determine source code authorship," *White Paper: Department of Computer Science, University of Wisconsin*, 2000.

[12] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum, "Human-level concept learning through probabilistic program induction," *Science*, vol. 350, no. 6266, pp. 1332–1338, 2015.

[13] S. Ravi and H. Larochelle, "Optimization as a model for few-shot learning," 2016.

[14] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[15] I. Loshchilov and F. Hutter, "Sgdr: Stochastic gradient descent with warm restarts," *arXiv preprint arXiv:1608.03983*, 2016.

[16] A. Krogh and J. A. Hertz, "A simple weight decay can improve generalization," in *Advances in neural information processing systems*, pp. 950–957, 1992.

[17] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.